

**DISTRIBUTED DATABASES**

A *database system* is a collection of information together with programs to manipulate that information. These programs present the user of the database with a structured interface to the database's information. We are interested in databases because some of them are distributed—they store information in many locations, often spanning great distances. Many advanced algorithms for organizing distributed systems have been developed as a result of the demand for coherent distributed database systems. Distributed databases strive to run programs at optimal locations (minimizing computation and communication costs) and to ensure the correctness of concurrent transactions. Distributed databases are among the most complex real distributed systems.

Database systems is a field in itself, suitable for study in advanced courses. We are not, of course, going to cover the field in its entirety. Instead, we focus on the points of particular interest to coordinated computing: the control of concurrency and distribution, and linguistic mechanisms for achieving this control. We introduce only a few formal database terms and assume no particular background in the field. In the first section of this chapter, we discuss concurrency control, replication, and failure and recovery mechanisms for distributed databases. In the second section, we introduce Argus, a language that incorporates notions of concurrency control, failure, and recovery into a programming language. We recommend that the reader who wishes to understand database systems in greater depth read one of the many texts on databases, such as Date [Date 81], Ullman [Ullman 82], or Wiederhold [Wiederhold 83]. The material in Section 17-1 is developed more thoroughly by Date [Date 83], Kohler [Kohler 81],

and Bernstein and Goodman [Bernstein 81]. Argus, described in Section 17-2, is the work of Barbara Liskov and her colleagues at M.I.T. [Liskov 83].

## 17-1 DATABASE CONTROL

Why distribute a database's data? Distribution greatly complicates constructing a correct database system. Two important motivations for the distribution of databases involve geography and reliability. Often, database users and data are themselves physically distributed. For example, banks keep databases of their customers' accounts. A bank with many dispersed offices may want local or regional processing centers to ease the transport of records and reduce the delay for inquiries. After all, most queries refer to local information and can be most economically processed locally. Nevertheless, the branches still need occasional, immediate access to the other regions' records. For example, if a customer from another region wants to cash a check, tellers must be able to quickly get that customer's records. Ideally, to the tellers, the customers' accounts should seem to be a monolithic whole. They should not be burdened with the details of where particular records are stored.

Second, distribution accompanied by replication can improve database reliability. *Replication* is storing the same information at several network locations. If the computer at the site of one copy of some particular datum is not working, the information can be obtained from another site with the same datum. This replication introduces the problem of ensuring that the multiple copies *appear* to be the same at all times.

Thus, two important aspects of distributed database systems are the location of the data and its consistent replication. A major design criterion for databases is that these two facets remain *transparent* to the database user. That is, the user should remain ignorant of which computer's data is being read, how far away that computer is, and, in general, the existence of other concurrent users of the database.

### Transactions and Atomicity

Databases organize information. Let us call the smallest information structure in a database a *record*. We imagine that primitive database operations read and write records. A typical banking database might have a record for each customer's checking account showing the account number, name, and balance; a typical airline reservation system database might have a record for each reservation showing the reserver, number of seats, and flight.

Some database queries or updates access only a single record and access that record only once. For example, a request for an account balance only needs to read the account record. However, most actions combine access to multiple records (find the total balance in a customer's accounts by summing the balance

in each), multiple varieties of access (find the balance in a customer's account, increment it by the amount of a deposit, and store the new balance), or a combination of reading and writing multiple records (transfer funds from one account to another).

Optimally, a database should be “correct” at all times. That is, it should always satisfy some consistency constraints. Thus, if a database stores both individual records of flight reservations and the total number of seats remaining on a flight, the total of all reservations and the remaining seats should remain constant (the number of seats on the plane). Unfortunately, this is not possible in general; the steps involved in updating the individual records leave the database temporarily inconsistent. For example, consider the following program which creates a reservation for a seat on flight *f* for customer *c*:

```
reserve (f: flight; c: customer) ≡
  avail := read (f, available_seats);
  if avail ≥ 1 then
    begin
      write (f, new_customer_record (c));    -- (1)
      write (f, available_seats, avail - 1)  -- (2)
    end
end
```

After line (1), the database is inconsistent. The total number of customer reservations and available seats exceeds the plane's capacity. Of course, the program quickly corrects this imbalance on line (2). Since inconsistent program steps are used in writing consistent programs, we cannot expect to have the database consistent at all times. Instead, we define a *transaction* to be an operation that takes the database from one consistent state to another consistent state. During any given transaction the database may be temporarily inconsistent. However, by the end of the transaction the database must be consistent again. For the remainder of our discussion, we assume that all user operations on our example database are transactions (if viewed in isolation from all other operations).

Databases often have many simultaneous users. For the database as a whole to preserve consistency, it is crucial that transactions be atomic—that transactions seem to be indivisible and instantaneous. Other processes should not become aware that a transaction is in progress. The transaction must also appear internally instantaneous—it should not see both the “before” and “after” of any other transaction.

In Section 3-2 we defined atomicity and presented an example of the inconsistency that can result when it is violated. It is helpful to review a similar example in our database context. Let us consider what happens if two different processes (transaction managers) simultaneously try to reserve a seat on flight 132 for customers Smith and Jones. We assume that there is a single remaining seat. As usual, each transaction manager has its own local temporary, *avail* (*avail'*).

Smith's transaction manager	Jones's transaction manager
<pre> avail := read (flight_132,               available_seats);       •       •       -- avail is 1, so write(flight_132,       new_customer_record (Smith));       •       • write(flight_132,       available_seats, 0)       •       • </pre>	<pre> • • avail' := read (flight_132,                available_seats);       -- avail' is 1, so       •       • write(flight_132,       new_customer_record (Jones));       •       • write(flight_132,       available_seats, 0) </pre>

Smith and Jones have both been given the last seat. The flight is now overbooked.\*

Of course, the problem is that the steps of the two transactions have been interleaved. Clearly, if we had executed the transactions sequentially (in either order), the database would have remained consistent. However, this is not to imply that transactions cannot be safely executed simultaneously. If Jones had been reserving a seat on flight 538, then the operations of his reservation could have been harmlessly interleaved with Smith's. We call a *schedule* an ordering of the elementary steps of a set of transactions. A schedule is *serial* if whole transactions are executed consecutively, without interleaving of the steps of other transactions. A schedule is *serializable* if its effect is equivalent to *some* serial schedule. In general, the goal in distributed databases is to produce a serializable schedule that maximizes potential concurrency—that is, to give each user the appearance of serial execution while still doing many things at the same time.

A slightly more general description of the previous conflict is that each transaction tried to first read and then update the same record. Schematically, this becomes

Time	Transaction P	Transaction Q
0	Read (r)	•
1	•	Read (r)
2	Write (r)	•
3	•	Write (r)

That is, at time 0, transaction P reads record *r*. At time 1, Q reads *r*. Each remembers what it has read. When P writes *r* at time 2, Q's knowledge of the

\* No, this is not how airlines really run their reservation systems.

database is now incorrect. Based on that incorrect value, its database update at time 3 can lead to an inconsistent state.

A *concurrency control mechanism* has three places that it could intervene to prevent this inconsistency:

- (a) At time 1, it could deny Q access to r, because P has already read it (and is likely to change it).
- (b) At time 2, it could refuse to let P write a new value for r, because Q is looking at the old value.
- (c) At time 3, it could refuse to let Q write a new value for r, because its update is based on the (now) incorrect value of r.

Many different algorithms for concurrency control have been developed. The two most important and recurring themes are locks and timestamps. Locks are used to ensure the synchronization of methods (a) and (b); timestamps for methods (b) and (c).

**Locks** Locking associates a *lock* with each database object. A transaction that *locks* an object makes that object inaccessible to other transactions. A transaction that tries to lock an object that is already locked must do one of three things: (1) wait for the object to unlock, (2) abort, or (3) cause the locking transaction to abort. Each of these possibilities has its costs. Waiting for the object to unlock reduces the potential concurrency in the system. More significantly, a pure waiting strategy can produce deadlock (processes waiting for each other). This deadlock must be detected and resolved by aborting one of the deadlocked transactions. Aborting a transaction involves restoring the objects the transaction has changed to their state before the transaction began (*rollback*). In order to be able to rollback, the system must maintain the old versions of records until it is sure that the transaction will not abort. Since the transaction itself is not in error (the problem was its coincidence with another transaction), it should be rerun.

Locking has the advantage of being able to ensure serializability. More specifically, Eswaran et al. [Eswaran 76] have shown that if every transaction is well-formed and two-phase then the resulting system is serializable. A transaction is *well-formed* if (1) it locks every object it accesses, (2) it does not lock any already-locked object, and (3) it eventually unlocks all its locks. A transaction is *two-phase* if it does all of its locking before any of its unlocking. Two-phase transactions pass through the phases of first acquiring locks and then releasing them.

The simplest kind of lock is an *exclusive (write, update) lock*. When a transaction places an exclusive lock on an object, it is the only transaction that can read or write that object. For example, a transaction should obtain an exclusive lock if it intends to read a record, compute a new value for that record, and

update the database with the new value. If another process wrote a different value into the record between the transaction's read and its write, that process's update would be overwritten and lost.

A transaction that only needs to read the database (and not update it) does not require such strong protection. For example, a transaction to compute the sum of a customer's accounts (but not store that sum in the database) would need to prevent a transfer of funds from one account to another during its computation. However, no harm results from another transaction reading the value of one of that customer's accounts while the first is processing. To handle this possibility, some systems provide *shared (read) locks*. A process that writes a database record must eventually obtain an exclusive lock on it. However, it can allow shared access until an update phase. Several processes can have shared locks on a record simultaneously. The system must still ensure that lock attempts fail if a transaction tries to get an exclusive lock and another transaction holds any lock on that record, or a transaction tries to get a shared lock and another transaction holds an exclusive lock on that record. Exclusive locks implement concurrency control mechanism (a) (see page 283), while shared locks implement mechanism (b).

Locking requires two steps, requesting the lock and granting it. Requests for a lock on a resource are directed at the resource's *lock manager*. Centralizing the lock management of all records at a single site simplifies the design of a distributed database and reduces the communication required to obtain and release locks. However, this scheme has two weaknesses: the lock manager is a potential communication bottleneck, and the entire system fails when the lock manager fails. Alternative structures involve making each database site the manager of its own data or distributing the management of record copies so that all the copies of any record are managed at a single site, but many sites are managers.

In a locking system, deadlock occurs when transactions are waiting for each other. Deadlock should, by now, be a familiar theme; avoiding the deadlock of five single-forked philosophers has been our goal in several example programs in earlier chapters. Operating systems often avoid deadlock by forcing processes to preclaim the resources they intend to use. That is, if a job requires two forks or five tape drives it must ask for them when it commences, not incrementally as it is running. Unfortunately, such deadlock avoidance techniques are inappropriate for databases. Transactions cannot know ahead of time which records they will access. On the contrary, a transaction often discovers the next search item only by analyzing the data in the last.

Since deadlock is hard to avoid in a locking database, deadlocks must be discovered and broken. The two primary mechanisms for resolving deadlocks are time-outs and deadlock detection. With time-outs, a transaction that has been waiting "too long" on a locked record aborts and restarts. This has the advantage of being easy to implement, but the disadvantage of being somewhat unfocused in its approach. If the waiting period is too short, some transactions may abort that are not deadlocked; if the waiting period is too long, the system

stays locked longer than necessary. Similarly, time-outs preclude fairness — a locked-out transaction can be timed-out repeatedly.

Deadlock detection steps above the actual locking processes and examines the *waiting-for* relation. In particular, if transaction A has a lock on record  $r$ , and transaction B is waiting for that lock, then B is *waiting for* A. A deadlock is a cycle of waiting transactions, with transaction  $A_0$  waiting for  $A_1$  waiting for ... waiting for  $A_n$  waiting for  $A_0$ . An algorithm for distributed deadlock detection requires potentially deadlocking transactions to report their waiting-for relationships to the deadlock detector. Practical distributed deadlock detection algorithms are a subject of current research.

**Timestamps** Locking is a pessimistic strategy. It assumes that transactions will conflict and acts to prevent that conflict. An alternative, “optimistic” approach is to assume that transactions will not conflict and to act only when they do. If this optimism is justified, the system enjoys increased concurrency, as transactions never wait on a lock. However, if this optimism is misplaced, the system performs redundant work, as actions that produce conflict are rerun.

The best way to implement the concurrency control mechanism (c) (see page 283) is with timestamps. (Timestamps can also be used to implement the mechanism (b).) A *timestamp* is a unique number associated with an object or event. This number can be thought of as the “time” the timestamp was issued. Although real times do not have to be used with timestamps, timestamps must be chosen from a strictly monotonic sequence. That is, if timestamp A is assigned after timestamp B, then A should be greater than B.\* Since timestamp systems never lock, the only way to prevent other transactions from seeing partial effects of a transaction is to make all the transaction’s changes simultaneously, at the end of its execution (at *commit time*). Later in this section we discuss the two-phase commit algorithm which ensures either that all of a transaction’s changes take effect or that none of them do.

*Conflict* occurs when a transaction tries to read a record written by a younger transaction (one with a larger timestamp) or tries to write a record that has already been seen or written by a younger transaction. (Thus, to detect conflict the system must maintain, for each record, the timestamps of the youngest transaction that has read that record and the youngest transaction that has successfully written it. Successful writing is committed writing.) Conflict can be resolved by

\* The astute reader may wonder how unique times are to be generated in a distributed system. One awkward way would be to have a centralized timestamp authority. This suffers from all the usual problems of introducing a central point to a distributed system. A better algorithm is to let each process issue timestamps that are the concatenation of that process’s real time and its unique identifier. (The real time must be the higher-order bits.) This ensures that no two timestamps are the same (provided each process’s clock “clicks” between issuing timestamps). Of course, process clocks may lose synchronization. This problem can be overcome by using the timestamp information of the communications each process receives to update its clock. An algorithm to ensure this synchronization is described by Lamport [Lamport 78].

waiting for a conflicting transaction to terminate (*wait*), or by aborting and restarting a transaction. A transaction can decide to restart itself. We call this action *dying*. Alternatively, a transaction could try to cause another transaction to restart. This is called *wounding*, because the wounded transaction may have already begun to commit its database changes and ought not be killed. If a wounded transaction has not already committed, it is aborted and restarted. The particular action taken is based on comparison of the timestamps of the conflicting transactions.

Two of the more prominent timestamp conflict-resolution algorithms are Wait-Die and Wound-Wait [Rosenkrantz 78]. In *Wait-Die*, if the requesting transaction is older it *waits*; otherwise it *dies*. In *Wound-Wait*, if the requesting transaction is older it *wounds*; otherwise it *waits*. (In each case, the first word describes the action of the requester if it is older, and the second word the action if it is younger.) Both protocols guarantee consistency and freedom from starvation. Both give priority to older transactions; in Wait-Die, an older transaction is spared death, while in Wound-Wait, an older transaction can attempt to preempt a younger one. Wait-Die has the undesirable property that a dying transaction can repeat the same conflict; Wound-Wait lacks this fault. On the other hand, a transaction under Wound-Wait can be interrupted at any time before committing, perhaps even during output, while a Wait-Die transaction only aborts at program-defined points (e.g., before accessing a data record).

**Summary** Locking and timestamps each have advantages. Locks require little space and (as Gray et al. [Gray 75] show) can be used with larger structures than single records. On the other hand, timestamp systems never deadlock and therefore do not have the problem of deadlock detection and resolution. The ultimate practical algorithms will probably combine some aspects of each, perhaps treating the serialization of reading and writing differently. Bernstein and Goodman discuss many of the possible combinations of control strategies, showing that almost all can be described as a selection from a regular framework [Bernstein 81].

## Replication

Data in a distributed database is, by definition, spread over several physical sites. One possible database organization is to store each record in one unique location. An alternative scheme is to replicate records, storing copies in several places. Replication can improve both performance and reliability. It improves performance by allowing a process to obtain data from a near copy instead of a distant original. Often, the major computational cost of a distributed query is the communication cost. (Of course, the ultimate near copy is one on the same processor as the requesting process.) Replication improves reliability because the failure of a site does not mean that a data set is inaccessible. Along with these advantages come the problems of propagating changes to all copies of a record

**Figure 17-1** Subnet partition through failures.

and ensuring that all copies appear to be the same at all times. In general, replicated systems should provide *replication transparency* — the user of the database should not discover which copy has been accessed (or even where or how many copies exist).

The possibility of individual site failures complicates the design of replicated systems. How can a system propagate updates to all copies of a record in the presence of failure? One technique is to update the working sites immediately and to keep a list of updates for the failed sites to read when they resume processing. When a site revives it needs to obtain its update list and perform the requested actions before processing new requests.

However, site failure is not the only kind of failure a system can experience. Both the failure of communication links and the failure of key network sites can break communication connectivity. These failures can partition the network into disjoint subnetworks (as in Figure 17-1) which are unable to communicate. Inconsistency may result if the subnetworks continue processing, updating their versions of the database. For example, two different subnetworks unable to synchronize their transactions might each give the last seat on our hypothetical flight to two different customers.

One replication algorithm that deals with the partitioning problem makes one copy of each record the *primary* copy. Different sites hold the primary copies of different records. The system first directs updates to the primary copy. When this copy has been changed, it propagates the updates to the rest of the database.

(The system must also ensure that if a transaction has written a record its subsequent read operations on that record see an updated copy.) Thus, if the network becomes partitioned, only the sites in the partition with the primary copy can update a record. However, the other sites can continue to read the record's old value. This scheme has the disadvantage that if the primary site for the record fails, the record cannot be updated at all.

Adiba and Lindsay propose an alternative approach to this problem, *database snapshots* [Adiba 80]. They suggest that many applications do not need exact, up-to-the-millisecond accurate views of the database. Instead, the system should periodically distribute copies of records (snapshots). Most applications will find that these copies satisfy their needs. When a program needs the exact value of a field, it can request it from the primary copy. Of course, updates are still directed at the primary copy.

**Failure, recovery, and two-phase commit** The design of a reliable system starts by specifying the meaning of reliability. This includes analyzing the kinds of errors expected and their probabilities, and defining an appropriate response to each. Kohler's *Computing Surveys* article is a good overview of failure, reliability, and recovery [Kohler 81]. Reliable database systems are built around stable storage that survives system failures (typically disk storage) and holds multiple copies of data records. In the discussion below we assume that the techniques described in that paper are followed and that disk storage is (adequately) reliable. That is, the database that writes a file on the disk will (with a high enough probability) be able to read back that same file.

Transactions are, by definition, atomic. If a transaction intends to change several different database records it must (to preserve its atomicity) either change them all or change none. Because individual sites can fail at any time, the problem of committing all the updates of a transaction together is more difficult for distributed systems. A transaction that does not commit at every site must rollback to the previous state and restart.

Reliability and the ability to rollback depend on keeping an *incremental log* of changes to the database on stable storage. This log must contain enough information to undo any unfinished transaction and to complete any committed transaction. Effectively, the incremental log keeps a record of system intentions. This record is used by the recovery procedure to determine what has been done and what remains to be done. A node runs its recovery procedure when it resumes processing after a failure.\*

\* By and large, the system that intends to do action  $X$  must first (1) write to stable storage that it intends to do  $X$  and then (2) do  $X$ . If it fails while writing its intentions, then the system is still in a consistent state;  $X$  has not been done at all. If it fails while doing  $X$ , then the recovery procedure can read the intention from stable storage, discover what part of the operation has completed, and continue or rollback. Systems must be sure not to first (1) do  $X$  and then (2) write to stable storage that  $X$  was done. If such a system fails after (1) but before (2) it cannot determine how to recover (or even if it needs to recover).

The standard algorithm for ensuring that the updates of a transaction either all commit or all abort is *two-phase commit* [Gray 79; Lampson 76]. With two-phase commit, every transaction has a commit coordinator. The *commit coordinator* is responsible for coordinating the other sites of the transaction (the cohorts), ensuring that every cohort either commits or every cohort aborts. As you might expect, two-phase commit has two phases. In the first phase, the commit coordinator alerts the cohorts that the transaction is nearing completion. If any cohort responds that it wants to abort the transaction (or does not respond at all), the commit coordinator sends all the cohorts abort messages. Otherwise, when all cohorts have responded that they are ready to commit, the commit coordinator starts the second phase, sending commit messages to the cohorts. At that point the transaction is committed. More specifically, the algorithms for the commit coordinator and cohorts are as follows:

### Phase one

#### **Commit coordinator**

- (1) Send a **prepare** message to every cohort.
- (2) If every cohort replies **ready** then proceed to Phase 2. If some cohort replies **quit** or some cohort does not respond (within the time-out interval) then:
  - (3) Write an **abort** entry in the log. Send an **abort** message to every cohort. Receive acknowledgments from each cohort and enter in the log. Repeat **abort** message to each cohort until it acknowledges.
  - (4) Terminate the transaction.

#### **Cohort**

- (1) Receive a **prepare** message from the commit coordinator.
- (2) Choose a response, **ready** or **quit**, and write that response on stable storage. Send that response to the commit coordinator.

### Phase two

#### **Commit coordinator**

- (1) Write a **commit** entry in the log. (*The transaction is now committed.*)
- (2) Send a **commit** message to each cohort.
- (3) Wait for an **acknowledge** response from each cohort. Enter acknowledgment in the log. Repeat **commit** message to each cohort until it acknowledges.
- (4) Enter **completed** in the log and terminate.

#### **Cohort**

- (1) Receive message from commit coordinator. If it is **commit** then:

(2a) Release locks and send commit coordinator **acknowledge**.

otherwise:

(2b) Undo actions, release locks, and send commit coordinator **acknowledge**.

If the commit coordinator fails during this protocol, its recovery procedure is as follows:

#### Recovery procedure

##### **Commit coordinator**

- (1) If it failed before writing the **commit** entry in the log, continue the commit coordinator at step (3) of Phase 1 (abort the transaction).
- (2) If it failed after writing the **completed** entry in the log, the transaction has been completed.
- (3) Otherwise, start the commit coordinator at step (2) of Phase 2.

## **17-2 ARGUS**

Atomic actions, recovery, and two-phase commit are clever inventions. This section is an overview of Argus, a language based on these ideas. Argus is concerned with manipulating and preserving long-lived, on-line data, such as the data of databases. Argus builds on the semantic architecture of remote procedure calls with processes whose communication structure resembles Distributed Processes (Section 13-2). Argus draws its syntactic and semantic foundation from CLU, a language developed around the ideas of data abstraction [Liskov 77]. It extends Distributed Processes in several ways, the most important of which is by making each external call part of an atomic action. This involves providing mechanisms to abort the calling structure if part of an atomic action fails and to recover from such failures. This recovery mechanism allows a software representation and resolution of hardware failures.

### **Actions and Atomicity**

The primitive processing object in Argus is a *guardian* — the “protector” of a resource, such as permanent data. Guardians have *handlers* (procedure entries) that can be called by other guardians. Guardians and handlers parallel the processes and procedure entries of Distributed Processes.

A guardian can have two kinds of storage, stable storage and volatile storage. Stable storage survives failures of the guardian (crashes of the guardian’s processor) while volatile storage is destroyed by such failures. Therefore, volatile storage can be used only to keep redundant information, such as a cache or an index into a data structure. Atomic actions transform stable storage from one

**Figure 17-2** A topaction-subaction tree.

consistent state to another. Argus guarantees that atomic actions have the appropriate effect on stable storage—all changes made by an atomic action are seen by other actions as having happened simultaneously; either all the changes of an atomic action take place or none do.

A primitive atomic activity in Argus is an *action*. An action completes by either committing or aborting. Of course, if an action aborts, the effect should be the same as if the action had never begun. Argus achieves serializable schedules by locking and (within the context of locking) keeping versions of stable storage. (However, as we shall see, versions in Argus are not as general a mechanism as the versions of timestamp systems.)

Argus has two kinds of actions, topactions and subactions (nested actions). A *topaction* is an attempt to get the stable storage of a system to achieve a new consistent state. If a topaction fails or aborts, all changes made by that topaction are discarded. While executing a topaction, a program can invoke other subactions.

*Subactions* play two important roles. The first is that concurrent activities are usually run in subactions. Subactions see the changes caused by their ancestor actions, but appear atomic to their sibling and cousin subactions. The other is that remote procedure calls (calls to the handlers of other processes) are always done in subactions. No subaction runs concurrently with its parent action.

Subactions can themselves have subactions. To another action, the subaction is invisibly part of the topaction. Thus, subaction invocation forms a tree structure, as in Figure 17-2. Like topactions, subactions can either commit or abort. Aborting a subaction does not abort its parent. Instead, the parent can detect the abort and respond accordingly. However, if a parent action aborts, all changes caused by its subactions are rolled back; the subactions are aborted, even if they had already reached their commit points.\*

\* The concept of subactions is not original with Argus. Earlier work on nested actions includes Davies [Davies 78], Gray et al. [Gray 81], and Reed [Reed 78]. Argus's novel idea is to make actions and subactions part of a programming language.

Argus uses a locking protocol to synchronize data access. It allows read locks and write locks. Issuing a write lock creates a new version of the locked object. The action then updates this version. If the action ultimately commits, this version becomes the “real” version of the data record and the old version is discarded. If the action ultimately aborts, the new version is discarded. These versions can be kept in volatile storage because they are temporary until their ancestor topaction commits. Argus uses two-phase commit to ensure that committed versions become permanent. Argus does not detect deadlocks. Instead, user programs can time-out topactions that appear deadlocked.

The introduction of subactions complicates the locking rules. Argus permits an action to obtain a read lock on an object if every holder of a write lock on that object is an ancestor action. An action can obtain a write lock if every holder of any lock on that object is an ancestor. For example, we imagine that the subaction structure shown in Figure 17-3 has evolved, with actions A through L having the indicated read (R) and write (W) locks on objects x, y, and z. Subaction J could now obtain a read lock on z [R(z)] and a write lock on x [W(x)]. However, J cannot obtain a read lock on y [R(y)], as C, a nonancestor, has a write lock; nor can it obtain a write lock on z [W(z)], as F, a nonancestor, has a read lock.

Since several actions could have write locks on a given object simultaneously, Argus must maintain multiple versions of objects. However, as actions with write locks form an ancestor chain and actions do not run concurrently with their descendants, only a single version of any object is active at any time. The versions of any object thus form a stack. All access is directed to the version at the top of the stack. When a subaction with any lock commits, its parent inherits its locks. When a subaction with a write lock commits, its version becomes the new version of its parent. Versions of aborting subactions are discarded.

**Figure 17-3** Locking rules for subactions.



```

create = proc (flag: bool) returns (cvt)  -- Cvt converts between an
                                          external, abstract
                                          representation and an
                                          internal, concrete one.

set = proc (af: cvt, flag: bool)
      af.flag := flag
end set

test = proc (af: cvt) returns (bool)
      return(af.flag)
end test

end atomic_flag

```

Argus introduces concurrency with the (**coenter**) statement, a form of **cobegin**. This statement initiates a parallel group of subactions or topactions. Argus provides an iterator (coroutine) construct in the **coenter** statement that generates elements and starts a process for each.

A child subaction of a **coenter** can abort its siblings. Thus, an Argus program seeking the value of a record from several sites of a distributed database could request them all in a single **coenter**, allowing the first subaction that succeeds to terminate the other requests.

In Distributed Processes, processes synchronize by setting the storage of a called process and testing it in guarded commands. Argus dispenses with this indirect mechanism. The primary use of synchronization is to ensure that data updates are consistent. But Argus's *raison d'être* is the consistent update of shared storage. Hence, calls to the handlers of a guardian are scheduled by the system independent of any explicit programmer control.

**Dining philosophers** We present a program to model the dining philosophers problem as our example of an Argus program. Unlike our other dining philosophers programs, this program does not have a centralized “room” object to prevent deadlocks. Instead (perhaps in keeping with more conventional eating habits), our philosophers try to pick up two forks. If a philosopher cannot get two forks soon enough, she backs off, dropping the forks to try again later. We ensure getting either two forks or none by making fork selection an atomic action. We run this action in parallel with an “alarm clock” action that aborts the pickup operation on time-out. Figure 17-4 shows the subaction structure of a single attempt to take two forks. This example uses the exception mechanism of Argus. A process that *signals* an exception transfers control to the closest (on the run-time stack) *exception handler* that handles that exception. Syntactically, the identifier **except** defines an exception handler, which lists in **when** clauses the various signals it handles.

**Figure 17-4** The subaction structure of claiming forks.

```

philosopher = guardian    -- the declaration of a process type
-- Argus guardian declarations begin with a specification part that indicates
  the names of both functions to create new instances of that type and
  that guardian's handlers (entry procedures).
is create    -- This function returns a new philosopher. A philosopher does
              not have any entry procedures.

-- These variables are on stable storage and survive the failure of the
  guardian's processor.
stable left, right : tableware    -- the two forks passed this guardian on its
                                  creation
stable taken      : atomic_flag -- set when this philosopher has her forks

recover    -- The guardian recovers from crashes by dropping any forks
  drop      she might have.
end        -- Argus eschews semicolons.

background
  while true do    -- Repeat ad nauseam.
    -- think
    take
    -- eat
    drop
  end
end

-- This function takes two forks and returns a new philosopher.
create = creator (l, r: tableware) returns (philosopher)
  left := l
  right := r
  taken := atomic_flag$create(false)
  return (self)
end create

```

```

take = proc ( )          -- to pick up both forks
while true do
  enter topaction      -- start a top-level action
  coenter              -- run the fork-getter and the alarm, in
                        -- parallel, as subactions
    action              -- fork-getter
      coenter
        action left.pickup
        action right.pickup
      end
    atomic_flag$set(taken, true)
    return             -- abort alarm, commit pickups
  action               -- the alarm process
    sleep (100 + random( ))
                        -- Delay some appropriate time. Make
                        -- this delay include a varying element.
    exit timeout -- abort fork-getting
  end
  except
    when timeout, already_taken:
    when failure (*):
  abort leave
end except when failure(*): end
sleep (200 + random( )) -- Wait awhile before trying again.
end
end take

drop = proc ( ) -- We drop the forks in a single topaction, dropping
                -- each in its own subaction. Either they both succeed or
                -- neither does.

while true do
  enter topaction
  begin
    if atomic_flag$test(taken)
      then coenter
        action
          left.putdown
          right.putdown
        end
      atomic_flag$set (taken, false)
    end
  return
end except when failure(*): end
abort leave

```

```

        end except when failure(*): end
    end
end drop
end philosopher

```

Forks return normally if free and picked up. They abort and signal an error if busy. The fork preserves its state in a single boolean variable.

```

tableware = guardian
  is create
  handles pickup, putdown

  stable busy: atomic_flag  -- Keep the state of the fork on stable storage.

  -- Forks have neither a recovery nor a background section.

  create = creator returns tableware
    busy := atomic_flag$create(false)
    return (self)
  end create

  pickup = handler ( ) signals (already_taken)
    if atomic_flag$test(busy) then
      signal already_taken
    end
    atomic_flag$set(busy, true)
  end pickup

  putdown = handler ( )
    if not atomic_flag$test(busy) then
      abort signal failure ("not picked up")
    end if
    atomic_flag$set(busy, false)
  end putdown
end tableware

```

The following statements create an array of five forks and an array of five philosophers, each indexed starting with 0. (Arrays in Argus are dynamic structures with no upper bounds.) The **for** loops pass the appropriate forks to the appropriate philosophers. Creating the guardians (calling their creators) sets them running.

```

forks : array[tableware] := array [tableware]$(0: ]
philos : array[philosopher] := array [philosopher]$(0: ]

```

```

for i: int in int$from_to(0, 4) do
    array[tableware]$addh(forks, tableware$create( ))
end

for i: int in int$from_to(0, 4) do
    array[philosopher]$addh(philos,
        philosopher$create(forks[i], forks[(i + 1)//5]))    -- //  $\equiv$  mod
end

```

## Perspective

Argus hones the procedures and procedure entries of conventional distributed programming languages into a tool directed precisely at the problems of distributed information storage systems. The other systems we have studied so far treat communication as the quantum unit of distribution. The programmer is responsible for building these quanta into a coherent algorithmic pattern. Argus identifies a class of conversations (atomic actions) involving many distributed processes and provides specific mechanisms for organizing these actions. A programmer can provide for one action in such a set to terminate the others or can insist that all must complete together. Argus is thus a heuristic approach to distributed computing—a recognition that coordinated problem solving requires tools that are both general and tuned to the issues of distribution.

## PROBLEMS

**17-1** Transaction A, with timestamp 2880, accesses database record R. If transaction B, with timestamp 2881, now tries to read record R and then update it, what happens? Assume that the system is using the conflict resolution scheme Wait-Die.

**17-2** Repeat the previous exercise for Wound-Wait.

**17-3** Transaction B, with timestamp 2881, accesses database record R. If transaction A, with timestamp 2880, now tries to read R and then update it, what happens? Assume that the system is using the conflict resolution scheme Wait-Die.

**17-4** Repeat the previous exercise for Wound-Wait.

**17-5** Could the synchronization mechanisms of timestamps be used to introduce “safe” side-effects to IAP (Chapter 12)?

**17-6** In Argus, how can the parent of a subaction abort while the subaction is still running?

**17-7** Why do the **sleep** commands in the dining philosophers program include a random value?

**17-8** Write an Argus program that books a multiple-segment airline trip by negotiating with the ticket reservation guardian of each airline. Make sure that your program reserves either all the seats for a trip or none of them.

**17-9** Often bids for houses are contingent on the sale of the bidder’s house. Such contingent bids expire after a specified time. Write an Argus program that mimics the bidding and sales of several house traders. Represent each house by a guardian that is receptive to certain bids. Confirm sales only after all contingencies have been removed.

**17-10** Program in Argus a majority-vote commit algorithm. In such a system, a transaction that commits at a majority of sites is deemed to have committed. Thus, the changes made

by that transaction are made permanent at those sites that have committed. Bernstein and Goodman's survey article discusses majority voting algorithms in detail [Bernstein 81].

## REFERENCES

- [Adiba 80] Adiba, M. E., and B. G. Lindsay, "Database Snapshots," *Proc. 6th Int. Conf. Very Large Data Bases*, Montreal (October 1980), pp. 86–91. Adiba and Lindsay propose that a database system take "snapshots" of the state of the database, distribute the snapshots, and use them for later processing. A process can use a local snapshot rather than requesting the information from a central repository.
- [Bernstein 81] Bernstein, P. A., and N. Goodman, "Concurrency Control in Distributed Database Systems," *Comput. Surv.*, vol. 13, no. 2 (June 1981), pp. 185–222. Bernstein and Goodman survey the state of concurrency control for distributed databases. They break concurrency control into two major subproblems, the synchronization of a reader and a writer, and the synchronization of two writers. They state several techniques for each subproblem and show how almost all practical algorithms for synchronization that have appeared in the literature are a selection of one of their techniques for each subproblem.
- [Date 81] Date, C. J., *An Introduction to Database Systems*, 3d ed., Addison-Wesley, Reading, Massachusetts (1981). Date's book is a good general introduction to database systems. It includes material on the nature of databases and the three major database models: the relational, hierarchical, and network models.
- [Date 83] Date, C. J., *An Introduction to Database Systems*, vol. II, Addison-Wesley, Reading, Massachusetts (1983). Date's second volume is perhaps the first book on advanced database principles. Of particular interest to coordinated computing are his chapters on concurrency, recovery, and distributed databases. He also covers material such as security, integrity, and database machines.
- [Davies 78] Davies, C. T., "Data Processing Spheres of Control," *IBM Syst. J.*, vol. 17, no. 2 (1978), pp. 179–198. Davies describes a database organization based on "spheres of control." These spheres effect nested transactions.
- [Eswaran 76] Eswaran, K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *CACM*, vol. 19, no. 11 (November 1976), pp. 624–633. This paper demonstrates that consistency requires two-phase algorithms—a consistent system cannot allow a transaction to acquire new locks after releasing old ones. Eswaran et al. then discuss the nature of locks, arguing that locks must control logical, not physical, sections of databases.
- [Gray 75] Gray, J. N., R. A. Lorie, and G. R. Putzolu, "Granularity of Locks in a Shared Data Base," *Proc. Int. Conf. Very Large Data Bases*, Framingham, Massachusetts (September 1975), pp. 428–451. In the text we specified that locks are associated with individual records. This paper describes an algorithm for locking sets of resources. The algorithm deals with records and class structures that are related by hierarchies and acyclic graphs. They introduce varieties of locks to achieve class-wide locking.
- [Gray 79] Gray, J. N., "Notes on Data Base Operating Systems," in R. Bayer, R. M. Graham, and G. Seegmuller (eds.), *Operating Systems: An Advanced Course*, Springer-Verlag, New York (1979), pp. 393–481. Gray focuses on the issues of recovery and locking in transaction systems. He presents a unified database design based on System R [Gray 81].
- [Gray 81] Gray, J., P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The Recovery Manager of the System R Database Manager," *Comput. Surv.*, vol. 13, no. 2 (June 1981), pp. 223–242. Gray et al. describe an experimental database system, "System R." System R is based on transactions, recovery protocols for dealing with failures, transaction logs, and saving system checkpoints.

- [**Kohler 81**] Kohler, W. H., “A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems,” *Comput. Surv.*, vol. 13, no. 2 (June 1981), pp. 149–184. Kohler surveys both concurrency control mechanisms for distributed databases and recovery mechanisms for database systems. He first describes locks, timestamps, and several other concurrency control mechanisms. He then deals with the general issue of recovery, which includes both the problems of secure storage and the requirements of recovery procedures.
- [**Lamport 78**] Lamport, L., “Time, Clocks, and the Ordering of Events in a Distributed System,” *CACM*, vol. 21, no. 7 (July 1978), pp. 558–565. Lamport shows that a process can keep its clock synchronized with the rest of the system if anytime it receives a timestamped communication from another process with a clock time greater than its own, it resets its own clock to that value.
- [**Lampson 76**] Lampson, B. W., and H. E. Sturgis, “Crash Recovery in a Distributed Storage System,” unnumbered technical report, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, California (1976). Lampson and Sturgis present an early version of a two-phase commit algorithm in this unpublished paper.
- [**Liskov 77**] Liskov, B., A. Snyder, R. R. Atkinson, and J. C. Schaffert, “Abstraction Mechanisms in CLU,” *CACM*, vol. 20, no. 8 (August 1977), pp. 564–576. CLU is a programming language based on abstraction—principally data abstraction. CLU provides the syntactic foundation for Argus.
- [**Liskov 82**] Liskov, B., “On Linguistic Support for Distributed Programs,” *IEEE Trans. Softw. Eng.*, vol. SE-8, no. 3 (May 1982), pp. 203–210. This paper describes a message-based precursor of Argus.
- [**Liskov 83**] Liskov, B., and R. Scheifler, “Guardians and Actions: Linguistic Support for Robust, Distributed Programs,” *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3 (July 1983), pp. 381–404. This paper is a preliminary description of Argus. Its major example is a distributed mail system that keeps track of mailboxes and forwards mail to the appropriate destination.
- [**Reed 78**] Reed, D. P., “Naming and Synchronization in a Decentralized Computer System,” Ph.D. dissertation, M.I.T., Cambridge, Massachusetts (1978). Reprinted as Technical Report TR-205, Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts. Reed introduces a concurrency control scheme based on keeping multiple versions of mutable objects and directing requests at the version with the appropriate timestamp.
- [**Rosenkrantz 78**] Rosenkrantz, D. J., R. E. Stearns, and P. M. Lewis, “System Level Concurrency Control for Distributed Database Systems,” *ACM Trans. Database Syst.*, vol. 3, no. 2 (June 1978), pp. 178–198. Rosenkrantz et al. present and compare several system-level concurrency control mechanisms based on timestamps. They introduce the Wait-Die and Wound-Wait algorithms.
- [**Ullman 82**] Ullman, J. D., *Principles of Database Systems*, 2d ed., Computer Science Press, Potomac, Maryland (1980). This is a text for an introductory database course. Ullman ties his development of database systems to other areas of computer science, such as theory and programming languages.
- [**Wiederhold 83**] Wiederhold, G., *Database Design*, 2d ed., McGraw-Hill, New York (1983). In contrast with other introductory books on database systems, Wiederhold concentrates less on the organization of particular databases and more on the quantitative aspects of database performance.
- [**Weihl 83**] Weihl, W., and B. Liskov, “Specification and Implementation of Resilient, Atomic Data Types,” *SIGPLAN Not.*, vol. 18, no. 6 (June 1983), pp. 53–64. Weihl and Liskov discuss atomic and resilient data types, particularly with respect to the emerging implementation of Argus.