Correctness Conditions for Highly Available Replicated Databases

Nancy Lynch, Massachusetts Institute of Technology
and Computer Corporation of America

Barbara Blaustein, Computer Corporation of America

Michael Siegel, Boston University and
Computer Corporation of America

Abstract

Correctness conditions are given which describe some of the properties exhibited by highly available distributed database systems such as the SHARD (System for Highly Available Replicated Data) system currently being developed at Computer Corporation of America. This system allows a database application to continue operation in the face of communication failures, including network partitions. A penalty is paid for this extra availability: the usual correctness conditions, serializability of transactions and preservation of integrity constraints, are not guaranteed. However, it is still possible to make interesting claims about the behavior of the system. The kinds of claims which can be proved include bounds on the costs of violation of integrity constraints, and fairness guarantees. In contrast to serializability's all-or-nothing character, this work has a "continuous" flavor: small changes in available information lead to small perturbations in correctness conditions.

This work is novel, because there has been very little previous success in stating interesting properties which are guaranteed by nonserializable systems.

Keywords and Phrases:

Databases, availability, nonserializable systems, integrity constraints, resource nonserializable systems, integrity constraints, resource allocation, cost bounds, fairness.

This work was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Air Force Systems Command at Rome Air Development Center under Contract No. F30602-84-C-0112. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advance Research Projects Agency or the U. S. Government. The work of the first author was also supported in part by the Office of Naval Research under Contract N00014-85-K-0168, by the Office of Army Research under Contract DAAG29-84-K-0058, by the National Science Foundation under Grant DCR-83-02391, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

1. Introduction

1.1. Background

In recent years, there has been extensive research on the design and theory of distributed databases. Nearly all of this work has been directed towards providing frameworks in which transactions can be processed concurrently, while preserving integrity constraints on the data. Many of the most important advances in distributed processing have arisen from this work, including the development of techniques based on locking and timestamps, and commit protocols. The work has led to elegant system designs, as well as to a very interesting theory.

It is apparent, however, that there is still a problem. The techniques developed in distributed database research have not yet been accepted by the commercial world to the extent that researchers might have hoped. In particular, airline reservation systems, banking systems and inventory control systems (applications which motivated much of the research), still do not rely on the general mechanisms developed by researchers. The problem may be fundamental to the general approach. The mechanisms developed in research guarantee preservation of integrity constraints, but they are inadequate for meeting stringent response time and availability requirements. This inadequacy seems to be an unavoidable result of strong requirements for synchronization among remote nodes.

Many applications of the sort mentioned above put a high premium on availability and fast performance, and in order to obtain these, they are willing to sacrifice something in the way of "correctness" or "data integrity". The research community has so far been unable to provide general frameworks which guarantee weaker correctness conditions as well as good performance and availability. As a result, practical systems development work for these applications is still based on ad hoc methods of concurrency control.

There is a need for system development work, as well as associated theory, to fill this gap. New frameworks are needed which guarantee good performance and availability, yet provide enough discipline on application programming so that useful correctness claims can be proved. When fast response time and high availability are required, it seems necessary to allow violations of integrity constraints to occur. In this case, traditional frameworks do not allow anything interesting to be proved about the behavior of the system. The difficult part of the problem is to guarantee interesting and useful correctness properties, even when integrity constraints are violated.

1.2. SHARD

The new SHARD (System for Highly Available Replicated Data) system under development at Computer Corporation of America (CCA) is designed to address the problems described above. It provides highly available distributed data processing in the face of communication failures (including network partitions). It does not guarantee serializability, nor does it preserve integrity constraints, but it does guarantee many practical and interesting properties of the database.

The reader is referred to [SBK] for a detailed description of the architecture of the SHARD system. Briefly, the main ideas are as follows. The network consists of a collection of nodes, each of which has a copy of the complete database. (Full replication is a simplifying assumption we have used for our initial prototype; many of our ideas seem extendible to the case of partial replication, but this extension remains to be made.) Replication allows transactions to be processed locally, thus reducing communication costs and delays, and providing high availability.

After a transaction is processed at its originating node, information about the transaction is broadcast reliably to all the other nodes for incorporation into the database copies at those nodes. The broadcast algorithm [GLBKSS] ensures that, barring permanent communication failures, every node will eventually receive information about every transaction. While the broadcast algorithm attempts to deliver information to all sites in as timely a manner as possible, communication and node failures can cause significant delays. Since nodes may continue to initiate transactions during communications failures - indeed, they may not even be aware that there is a failure somewhere in the network - these delays mean that transactions may run against out-of-date database states.

When a node receives new information about a transaction, no matter when the transaction was initiated, this information must be merged into the node's copy of the database; this merging must be done consistently at all nodes, to maintain mutual consistency. The following mechanism is used to guarantee consistent merging. Transactions are totally ordered by a globally-unique timestamp assignment (such as one based on local timestamps with node identifiers used for tiebreaking), and each node uses this total ordering to determine how to merge information about different transactions. Because all nodes order the transactions in the same way, they will agree on the result of merging identical sets of transactions. Also, at all times during execution, each node's copy of the database always reflects the effects of all the transactions known to that node, as if they were run according to the global timestamp order.

Since messages about different transactions could arrive at a single node out of timestamp order, keeping the copy correct entails frequent undoing and redoing of transactions. The SHARD system uses an undo-redo strategy in lieu of any other inter-node concurrency control mechanism. This strategy allows the nodes to achieve mutual consistency without relying on extra network communication. There are several implementation ideas which reduce the amount of undoing and redoing that is actually necessary; some of these are discussed in [BK,SKS].

Problems arise with the simple scheme described so far in its interactions with the external world. Certain transactions will trigger external actions. For example, in an airline reservation system, a booking transaction might determine that there are available seats on a flight, and might cause a passenger to be informed that he has been assigned a seat. Although the transaction is run at different nodes, and possibly undone and redone many times, the external action should only occur once - at the transaction's origin node, when the transaction is initiated.

When a transaction is rerun at a node, it may be necessary to undo all its effects before redoing it starting from a different database state. This requirement is a serious problem for transactions which trigger external actions: it is not possible for the system to undo an external action. Moreover, when the transaction is redone, it might not choose to trigger the same external action. In an airline reservation system, a booking transaction might decide to inform a passenger of an available seat when the transaction is initiated. However, if this booking transaction is undone and then redone from a database state in which there do not appear to be any available seats, it would not grant the seat. Thus, after the undo and redo, the database would not record the fact that the passenger had been granted a seat, even though the passenger has actually been informed that a seat has been granted. This situation produces an inconsistency between the information in the database and the information sent to the passenger. We would like to avoid this kind of inconsistency.

Thus, we find it useful to limit the interaction of transactions with the external world, by imposing some extra structure on the transactions. We require that all transactions be divided into two parts: a "decision", which may read data and trigger external actions, but may not modify the database, and an "update", which may read and write the database but may not trigger external actions.

The decision part of a transaction is invoked only when the transaction is initiated. This part of the transaction may interact with the user, giving some indication of the likely outcome of the completed transaction. The results returned by the decision determine an update, which is then broadcast to all the nodes to be merged into all the copies of the database. Only the update is broadcast to the other nodes. The update is the part of the transaction that may be undone and redone; the decision is executed only once. Since the decision involves no changes to the database, just broadcasting the update is enough to insure mutual consistency of the database copies.

In the example described earlier, the decision part of the booking transaction could read the database at the local (initiating) node and determine whether there appear to be available seats. If there are, the decision would inform the requesting passenger that he has been granted a seat, and would also cause the system to invoke an update that writes the reservation into the database. When the update is received by the other nodes, the reservation is also entered into their copies of the database. Thus, every node would correctly record the fact that the passenger was granted a seat.

Because of the distribution, and because of the possible need for undo and redo, the update part of the booking transaction may execute many times, possibly from different database states. No matter what state it is executed from, the update records the facts that the seat was assigned and the passenger was informed of the assignment. This update records the facts correctly even if it is executed from a state from which a booking transaction run in its entirety would not choose to grant the passenger a seat.

Because decisions are made with incomplete information about the updates of preceding transactions, it is possible that the database could reach an undesirable state, e.g. a state in which a flight is overbooked. However, users or application programmers could monitor the database with additional "compensating" transactions, which invoke appropriate corrective actions. In this example, a transaction might check for overbooking, and decide on a particular passenger to unseat. The decision part of this transaction would inform the passenger that his reservation has been rescinded. The update would just record, in the database, the fact that the particular passenger has been unseated. Of course, applications should be designed to avoid an excessive amount of compensation. The correctness conditions described in this paper should help to provide application designers with guidelines for coping with these and other problems caused by a lack of serializability.

A preliminary design for SHARD has been completed, and is documented in [BK,GLBKSS,S,SBK,SKS]. Also, a prototype implementation is completed.

1.3. Correctness Conditions

The SHARD system can be implemented efficiently, and seems capable of expressing the kinds of transaction behavior actually used in commercial systems. However, if the system is going to be widely used, it should be possible to make precise claims about its behavior. This paper provides a formal setting in which such claims can be made, and uses that framework to prove some interesting claims about SHARD's behavior.

It should be clear that SHARD does not guarantee serializability of complete transactions. It does guarantee serializability of the update parts of transactions, but that condition by itself does not say very much. We believe that we can say more about what is guaranteed by such a system than just what we can conclude from its weak serializability properties.

We take our cue from some of the intended applications of the system, such as airline reservations, banking, and inventory control. These exemplify different kinds of resource allocation applications. In all these cases, there are natural integrity constraints which one would want to define; these are usually expressed as predicates on the database states. In resource allocation applications, one useful integrity constraint would be that the number of allocated resources be no greater than the number of available resources. Another would be that the number of allocated resources be no less than the number of available resources, provided there are enough requests for resources. Both of these conditions are described by predicates on the database state.

However, one can go further: there is often a "cost" associated with violations of an integrity constraint, which can be expressed as a function of the database state. In resource allocation applications, the cost of over-allocation might be some number which is proportional to the excess of the number of allocated resources over the number of available resources. The cost of unnecessary under-allocation might be proportional to the minimum of the number of unsatisfied requests, and the excess of the number of available resources over the number of allocated resources. Each of the applications listed has its own particular cost functions, characteristic of that application. In each case, it is desirable to keep the costs as low as possible.

Thus, one kind of property we would like to prove is a bound on the cost of violations of integrity constraints. Results of the form "With absolute certainty, the cost remains at most c." would be unreasonably strong in our setting, because of the uncertainty that arises from delays and failures. Rather, it seems much more appropriate to prove results of the form "With probability p, the cost remains at most c." Results of this form would be very useful to the application designer, since they would allow him to adjust his design in such a way as to lower the expected cost bound.

We believe that results of this form, are most conveniently proved in two parts: (1) conditional results of the form "If certain conditions hold, then the cost remains at most c.", and (2) probability distribution information describing the probability that the conditions hold. Most often, the conditions mentioned in (1) will be parametrized, e.g. "When each transaction is initially executed, the database state includes the effects of all but at most k of certain kinds of preceding transactions." Similarly, the cost mentioned in the conclusion of (1) will be parametrized. Thus, results of type (1) will usually be a class of related results, giving cost bounds for a range of quantitatively different assumptions about system operation. The probability distribution information in (2) will be obtained by an independent analysis, using information such as delay characteristics of the message system, and expected rates of transaction processing. It should be relatively easy to combine the information in (1) and (2) to get probabilistic statements of the kind we want. In this paper, we do not carry out the probabilistic analysis required in (2), but instead focus on the parametrized conditional claims in (1).

Thus, we obtain results of the form "If each transaction "sees" all but at most k of certain kinds of preceding transactions, then the cost remains at most c(k)." Such cost bounds limit the damage which can be caused when transactions operate with a bounded amount of missing information. The cost bounds we obtain are, in general, intuitively natural, rather than extremely surprising; our main contribution lies in the fact that we can actually formulate and prove the intuitive claims. Previously, no claims at all could be made when information about any transactions was missing. We can make such claims, and our claims become stronger (i.e. the integrity constraints are better preserved) when information is more complete (i.e. when execution is closer to being serializable). In contrast to serializability's all-or-nothing character, our work has a "continuous" flavor: small changes in available information lead to small perturbations in integrity constraints.

The question of how the costs get defined still remains to be addressed. Assignment of costs is something that must be done by application programmers, who understand the impact of database behavior on the organization using the system. It is likely that the cost assignment procedure will be complex and approximate. Nevertheless, it appears to be what is currently used by organizations, implicitly, in evaluating the acceptability of database system behavior. Therefore, it seems that such cost assignments should play an important role in evaluating database behavior.

Another kind of property which is of interest for resource-allocation applications is "fairness". Fairness properties describe conditions under which a particular request is guaranteed to be granted, or guaranteed not to be granted. They also deal with relative priority of different requests in obtaining resources. While FIFO order might be an appropriate fairness condition in a serializable system, weaker fairness conditions are more appropriate in the SHARD setting, and are still of interest.

In this paper, we begin by providing the basic definitions and vocabulary for discussing the operation of systems of this type. Then, following the usual organization in traditional concurrency control theory, we study the correctness conditions in two groups. First, we examine conditions which can be guaranteed by the system alone (analogous to serializability). The system does guarantee to run transactions in some total order. But whereas serializability would guarantee that each transaction has total information about the effects of the preceding transactions, the SHARD system only guarantees that each transaction has partial information about the preceding transactions. Second, we examine conditions which can be guaranteed by the transactions (analogous to preservation of integrity constraints). Transactions might be required not just to preserve integrity, but also to improve or restore integrity. These two kinds of conditions, those guaranteed by the system and those guaranteed by the transactions, can be combined to allow proof of interesting properties (cost bounds and fairness) for a running application.

We describe our properties and carry out our proofs in the context of a simple prototypical resource allocation example. We believe that this example contains many of the elements common to the class of applications for which SHARD is suited. The types of conditions stated and the techniques for proving their correctness appear likely to extend to the other applications. Wherever possible, we state conditions and describe proofs in a general way, so that they will be directly applicable to other applications.

Related work includes several other papers which weaken serializability in various ways [FM, AM, G, B, for example]. Other work that seems related to the SHARD approach, although in a very different context, is the work on "virtual time" [J].

The rest of the paper is organized as follows. In Section 2, we describe our database model. In Section 3, we describe conditions that can be guaranteed by the system alone. In Section 4, we describe conditions that can be guaranteed by the transactions alone. In Section 5, we prove some interesting cost bound and fairness properties for the example resource allocation system. These properties are consequences of both the conditions guaranteed by the system and those guaranteed by the transactions. In Section 6, we present our conclusions.

2. Database Model

This section includes formal definitions of database states, integrity constraints, and transactions.

One goal of the SHARD design is to keep the distribution and replication of data hidden from the application. In particular, we attempt to avoid explicit mention of distribution and replication in our correctness conditions. Our general approach is analogous to the usual approach for describing correctness of distributed databases [BG, for example]. In the usual approach, correctness of a distributed database requires that the distributed database give the appearance of a centralized, serial database. In our case, the database will not appear to be serial, but will still appear to be centralized.

In other database research, certain consistency conditions, called "integrity constraints," are given for the database states. These conditions fit into our model in two ways. The most fundamental are modelled as "well-formedness" conditions; we will require that transactions always preserve these. The other consistency conditions, which we call "integrity constraints," represent desirable conditions, but we do not assume that they are preserved at all times. To measure how far a database state is from satisfying the integrity constraints, we impose cost measures on the states with respect to each constraint, where a greater cost indicates that the state is further from satisfying the constraint. One goal of SHARD is to minimize the cost of states that arise during an execution.

Our transactions are composed of two parts, a "decision part" and an "update." As described in the Introduction, the decision part reads data and may interact with the external world, but does not modify the database. The results returned by the decision part determine an update, which can read and write the database, but does not directly interact with the external world.

In addition to providing general definitions in this section, we also define an airline reservation example, with four transactions. This example will be used throughout the rest of the paper.

2.1. States

The database has a set S of possible *database states*, among which a particular *initial state* s_0 is distinguished. There might be some additional structure on the database; for example, it might be composed of a collection of *objects*, where a state would consist of a *value* for each object. In case X is an object, we let *X(s)* denote the value of object X in database state s.

Among the database states, there may be some which fail to satisfy some fundamental consistency conditions, and we will generally want to omit them entirely from consideration. Therefore, we designate certain of the database states as *well-formed*. We assume that the initial state is well-formed.

Example:

Fly-by-Night Airlines is a little-known airline company which has exactly one scheduled flight, Flight 1. Flight 1 is scheduled to take off next Jan. 1 and will take its lucky 100 passengers from Boston to an idyllic resort in the Caribbean.

A database state consists of the following objects:

- ASSIGNED−LIST, a finite ordered list of people who have been notified that they have seats on Flight 1, and

- WAIT−LIST, a finite ordered list of people who have requested seats on Flight 1, but do not have assigned seats.

The initial state has both lists empty. The well-formed states are those which satisfy the fundamental consistency condition that ASSIGNED−LIST and WAIT−LIST must contain disjoint sets of people.

We use the notation AL(s) as a shorthand for |ASSIGNED−LIST(s)|, the number of people on the assigned list in state s; similarly, we use WL(s) for |WAIT−LIST(s)|. We will sometimes refer to AL and WL as if they were objects themselves; they are similar to objects, in that they have values in every database state. However, those values are always derived from the values of the "real" objects, ASSIGNED−LIST and WAIT−LIST.

2.2. Integrity Constraints

For us, "integrity constraints" represent desirable conditions, but we do not assume that they are preserved at all times. Since integrity constraints are not always preserved, we find it useful to measure how far a database state is from satisfying the integrity constraints. In order to do this, we impose nonnegative real-valued cost measures on the states with respect to each constraint, where a greater cost indicates that the state is further from satisfying the constraint. A cost of zero indicates that the constraint is satisfied. The total cost of a state is the sum of the costs associated with all the constraints. One goal of SHARD is to minimize the cost of states that arise during an execution.

More precisely, we assume a finite collection of integrity constraints, indexed by the set I. Let *cost(s,i)* denote the cost of database state s which is attributed to a violation of integrity constraint i. The cost of s, *cost(s)*, is then defined as $\Sigma_{i \in I}$cost(s,i)

We use the notation X ∸ Y to denote max(X-Y,0).

Example:

In the Fly-By-Night airline reservation system, there are two integrity constraints in addition to the well-formedness condition already described.

Integrity Constraint 1: Overbooking should not occur.

Formally, this says that $AL \leq 100$. While this condition is certainly desirable, we do not expect that it will always hold. If Flight 1 is overbooked, the cost to Fly-by-Night Airlines is approximately $900 per overbooked passenger. (This cost covers the price of a first-class ticket on an alternative flight, plus hotel accomodations for a week in the Caribbean.) Thus, we define cost(s,1), the cost of state s which is attributed to violating constraint 1, to be 900 (AL(s) ∸ 100).

Integrity Constraint 2: Underbooking should not occur, if it is avoidable.

Formally, this says that either $AL \geq 100$ or else $WL = 0$. That is, either all the seats on Flight 1 are assigned or else there are no waitlisted passengers. If Flight 1 is unnecessarily underbooked, the cost to the airline company is approximately $300 for each waitlisted passenger who could have been assigned a seat. (This is the missed profit.) Thus, we define cost(s,2), the cost of state s which is attributed to violating constraint 2, to be 300 min(100 ∸ AL(s), WL(s)).

The assignment of costs to database states, for violation of particular integrity constraints, is a part of application design. In practice, it might not always be obvious how to assign such costs. It is possible that the system could help the application designers, by providing a framework in which the designers could determine appropriate cost functions. Cost functions often summarize other information which the application designers might find it easier to think about. For instance, in many interesting cases (such as the airline reservation system), the data is numerical, and the cost functions have some simple (e.g., linear) relationship to the data values. Perhaps patterns such as this one could be incorporated into a language for describing cost assignments. Systematizing cost assignments is a subject for future research.

2.3. Transactions

In this subsection, we describe the structure of transactions. As noted earlier, our transactions are composed of two parts, a "decision part" and an "update". The decision part reads data and may interact with the external world, but does not modify the database. The results returned by the decision part determine an update, which can read and write the database, but does not directly interact with the external world.

Formally, an *update* is any mapping from S to S which preserves well-formedness. Let $\mathcal{A}$ denote the set of updates. Let $\mathcal{E}$ denote the set of external actions. A transaction T consists of a *decision part* D_T which is a mapping from the state set S to $\mathcal{A} \times \mathcal{P}(\mathcal{E})$. For any database state s, $D_T(s)$ is a pair consisting of the *update* which is invoked when T is run from s, and the set of external actions triggered by T when T is run from s. Where no confusion is likely, we will sometimes write $D_T(s)$ to denote just the update, ignoring the external actions.

A transaction is designed to execute nonatomically; it "observes" some state of the database when it is initially run, but then later it transforms other, possibly different, states. The observation of the database takes place in the decision part, and the state transformation in the update part. Each of these two parts is intended to be carried out atomically. The state that a transaction observes is to be thought of as embodying partial information about past updates, such as the information known at the local site at the time the transaction is first executed. This partial information is used to decide on the new update to be generated.

Example:

The airline reservation system has only four transactions: a REQUEST for a seat which puts the passenger on the waiting list, a CANCEL transaction, a MOVE−UP transaction which moves a waitlisted passenger to the assigned list, and a corresponding MOVE−DOWN transaction which moves an assigned passenger back to the waiting list. Note that we are departing slightly from the example discussed in the Introduction: the effects of the booking transaction described there are achieved by a combination of a REQUEST transaction and a MOVE−UP transaction.

The four transactions are as follows:

(1) REQUEST(P), where P is a person

This transaction is described by the following program.

```
Decision: TRUE
  Action:
      if P is not on WAIT—LIST and P is not on
              ASSIGNED—LIST
          then add P to end of WAIT—LIST
```

This program is to be interpreted as follows. For any state s, the decision mapping $D_{REQUEST(P)}$ triggers no external action and invokes the same update A. A operates on any state s' by adding P to the WAIT−LIST provided that P is not already on either the WAIT−LIST or the ASSIGNED−LIST, in s'. In case P is on either list in s', A does nothing. We refer to the unique update A invoked by the REQUEST(P) transaction, as the *request(P)* update.

(2) CANCEL(P), where P is a person

This is described by the following program.

```
Decision: TRUE
  Action:
      if P is on WAIT—LIST
          then remove P from WAIT—LIST
      if P is on ASSIGNED—LIST
          then remove P from ASSIGNED—LIST
```

Again, from any state s, the decision mapping always yields the same update. This update, from any state s', removes P from any list on which it happens to appear. If P is not on either list, the update does nothing. We refer to the unique update invoked by the CANCEL(P) transaction, as the *cancel(P)* update.

The decision parts of the REQUEST and CANCEL transactions do not perform any interesting work: they always invoke the same update, and trigger no external actions. On the other hand, the following two transactions have decision parts that invoke different updates in different situations, and they sometimes trigger external actions.

(3) MOVE−UP

```
Decision: AL < 100 and WL > 0 and P is the first person
              on WAIT—LIST
  External event: inform P that P is now assigned a seat
  Action:
      if P is on WAIT—LIST
          then
            [remove P from WAIT—LIST
            add P to end of ASSIGNED—LIST]
```

Here, the decision part, running from state s, tests to see whether there is room on the ASSIGNED−LIST and a person waiting to be assigned. If not, no action is taken. If so, the decision part selects a particular person P (the first on the WAIT−LIST in state s) to be moved up from the WAIT−LIST to the ASSIGNED−LIST. A message is sent to P, and the update is parametrized by P. From any state s', the update moves P from the waiting list to the end of the assigned list, provided that P is actually on the waiting list in s'. Otherwise (i.e. if P is already on the assigned list, or P is on neither list), no change occurs. We refer to the update generated by the MOVE−UP transaction when it selects person P as the *move−up(P)* update.

(4) MOVE−DOWN

```
Decision: AL > 100 and P is the last person on
              ASSIGNED—LIST
  External event: inform P that P is now waitlisted
  Action:
      if P is on ASSIGNED—LIST
          then
            [remove P from ASSIGNED—LIST
            add P to end of WAIT—LIST]
```

The meaning of this transaction is symmetric with the preceding one. We refer to the update invoked by the MOVE−DOWN transaction when it selects person P as the *move−down(P)* update.

It is clear that all the updates, for all four transactions, preserve well-formedness, as required.

Note that each of the last two transactions contains two conditionals. The two conditionals play different roles. The first conditional in each case is used to decide which update and external actions will occur. The second is part of the execution of the update. Also note that the transactions are designed to observe the database state more than once. For example, in the MOVE−DOWN transaction, the transaction looks at ASSIGNED−LIST in one state s in order to attempt to select a person P to move down. Then whenever the move-down(P) update is executed, it looks at ASSIGNED−LIST in another state s' to determine whether to actually move P.

We consider this airline reservation system to be a prototype of a much more general class of resource allocation systems. It seems that practically all resource allocation systems must have operations of the four kinds described above: operations that request resources and cancel those requests, as well as operations that allocate and deallocate the resources. Those operations will behave in somewhat different ways for each application. Here, to be specific, we have made a particular set of choices, but we expect that many of the ideas in this paper will carry over to other resource allocation systems.

We introduce some additional notation which will be useful later for describing transactions. If the first component of $D_T(s)$ is an update which maps state s' to state s'', we will write T(s,s') = s''. If T(s,s') = s'', it means that if T is initially run from state s, it causes the system to invoke an update which, if it is ever run from state s', will produce state s''.

3. Conditions Guaranteed by the System

This section describes conditions that can be guaranteed by the system alone, i.e. conditions on how the system will run the transactions. Later, in Section 4, we describe conditions that can be guaranteed by the transactions alone. Then in Section 5, we combine these two kinds of conditions to prove properties of an application (the Fly-by-Night Airline Reservation System) running on the system.

This approach is roughly analogous to the usual approach in ordinary concurrency control theory. There, the serializability condition (which can be guaranteed by the system alone) is combined with the condition that individual transactions preserve integrity (which can be guaranteed by the transactions alone), to conclude that reachable database states all satisfy the integrity constraints.

The first subsection formally describes the basic guarantees made by SHARD about the way in which transactions are run. SHARD guarantees that there is some serial order for the transactions which it runs. The system does not guarantee serializability of the transactions in this order, but it does guarantee that each transaction "sees" the result of some subsequence of the preceding transactions. While this condition is fundamental to the semantics of the system, it is too weak to allow proof of interesting properties.

The second subsection contains refinements of the basic condition. Examples of these refinements are transitivity and some specific requirements on the subsequences of transactions seen by certain other transactions. The third subsection describes implementation issues. It shows how SHARD and similar systems can guarantee the conditions described in the other two subsections.

3.1. The Prefix Subsequence Condition

The system guarantees that there is some serial order for the transactions which it runs, and that each transaction "sees" the result of some subsequence of the preceding transactions in this serial order. We state this condition more formally below.

If s is any sequence, we write s_i to denote the ith element of s. An *execution* of a set of transaction instances, consists of a serial ordering T for the transaction instances, together with a sequence A of updates, a sequence E of sets of external actions, a sequence $\mathcal{T}$ of finite sequences of integers, and two sequences, s and t, of database states. An execution is required to satisfy the following conditions.

1. For $i \geq 1$, $\mathcal{T}_i$ is a subsequence of the prefix sequence $\{1,...,i-1\}$.

2. For $i \geq 0$, t_i is the state obtained by applying the sequence of updates designated by $\mathcal{T}_{i+1}$ to the initial database state s_0. That is, $t_i = A_{i_k}(...A_{i_1}(s_0))$, where $\mathcal{T}_{i+1} = \{i_1,...,i_k\}$.

3. For $i \geq 1$, $(A_i,E_i) = D_{T_i}(t_{i-1})$.

4. For $i \geq 0$, each s_i is the state obtained by applying the sequence of updates $A_1,...,A_i$, to s_0. That is, $s_i = A_i(...A_1(s_0))$.

These conditions mean the following. (1) says that each transaction T_i has a corresponding subsequence $\mathcal{T}_i$ of its prefix of preceding transactions; these are the preceding transactions that it "sees". (2) says that each state t_i describes the effects of the updates of T_{i+1}'s prefix subsequence; it is the state of the database which T_{i+1} "sees" when its decision part is run. (3) says that the update and external actions produced by T_i are determined by its observed state t_{i-1}. Finally, (4) says that the states s_i describe the actual effect (not necessarily observable by any of the transactions) of running the complete sequence of updates generated by all transactions through T_i.

The system guarantees to simulate (in some sense which we do not specify here) executions of those transactions which are submitted to it. In particular, it guarantees that the external actions described by sequence E are actually performed.

We say that the *apparent state before* transaction T_{i+1} is t_i, and that the *apparent state after* transaction T_{i+1} is state $T_{i+1}(t_i,t_i)$. Also, the *actual state before* transaction T_{i+1} is s_i, and the *actual state after* transaction T_{i+1} is state $s_{i+1} = T_{i+1}(t_i,s_i)$. We extend this notation to nonempty consecutive sequences of transactions in place of single transactions: the apparent and actual states before the sequence are just the apparent and actual states, respectively, before the first transaction in the sequence, while the apparent and actual states after the sequence are just the apparent and actual states, respectively, after the last transaction in the sequence. We say that each of the s_i is *reachable* from s_0 in the given execution. We call the state s_{i-1} the *complete prefix state* for T_i in the given execution.

Let $\mathcal{U} = \{i,i+1,...\}$ be a sequence of consecutive indices. Then $\mathcal{U}$ is said to be *atomic* in an execution provided that the following hold. (a) Each U_j, $j \in \mathcal{U}$, includes each of the other transactions U_k, $k \in \mathcal{U}$, $k < j$, in its prefix subsequence, and (b) all transactions U_j, $j \in \mathcal{U}$, have the same subset of the transactions with indices less than i in their prefix subsequences. Atomicity describes the running of several consecutive transactions without allowing new information about the database to intervene.

The prefix subsequence condition only guarantees that each transaction sees the result of some subsequence of its prefix. This condition does not rule out trivial solutions, such as every transaction seeing the initial database state (the result of the empty subsequence). In order to insure useful behavior, we would like the system to allow transactions to see prefixes which are as large as possible. Some refinements of the prefix subsequence condition designed to insure large prefixes are discussed in the following subsection.

Example:

This example shows an execution of the transactions from the airline reservation system, acting non-serializably, but according to the prefix subsequence condition specified above. The left-hand column lists the successive T_i, while the right-hand column lists the corresponding A_i.

T	A
REQUEST(P1)	request(P1)
MOVE—UP	move—up(P1)
REQUEST(P2)	request(P2)
MOVE—UP	move—up(P2)
...	
REQUEST(P102)	request(P102)
MOVE—UP	move—up(P102)
MOVE—DOWN	move—down(P101)
CANCEL(P1)	cancel(P1)

This execution can be obtained by having all the requests, the first 100 MOVE−UP transactions, and the cancellation operate seeing complete prefixes. The next two MOVE−UP transactions operate with incomplete prefixes. The first sees the results of the first 99 REQUESTS and MOVE−UPS, plus the REQUEST for P101, while the second sees the results of the first 99 REQUESTS and MOVE−UPS, plus the REQUEST for P102. Since each observes a state with only 99 people on the assigned list, each chooses to move a person up. Similarly, the MOVE−DOWN operates with an incomplete prefix. It sees the results of the first 202 transactions only, but not the results of the two transactions involving P102. Thus, it sees the assigned list with 101 people, and moves P101, the person it observes to be last, down.

Now consider the successive reachable states s_i. The state after the first 204 transactions, s_{204}, has 102 people on the assigned list, in numerical order, and no one on the waiting list. After the MOVE−DOWN, s_{205} has P101 on the waiting list and P1,P2,...,P100,P102 in order on the assigned list. The final cancellation then leaves the assigned list with exactly 100 passengers: P2,...,P100,P102.

This execution differs from a serializable execution in at least two ways. First, there is a reachable state (s_{204}) for which the overbooking cost is nonzero. Second, the execution is not entirely "fair" in that P102 requests a seat after P101 (and his request is processed after P101's), but P102 is allowed to remain on the assigned list while P101 is moved down.

Notice that there is a danger of "thrashing" in this system. If a MOVE−UP transaction does not see a previous request and corresponding MOVE−UP, say for person P, it may move another person Q to the assigned list. A later MOVE−DOWN transaction might operate with a complete prefix, observe an overbooking, and move Q down. Another MOVE−UP might then execute, seeing the move−down(Q) update, but still not seeing the updates missed by the previous MOVE−UP; it may then reassign Q. A later MOVE−DOWN might then move Q back down, and so on. This kind of thrashing is very undesirable, not just because of its obvious inefficiency, but because of the external effects of the conflicting transactions.

3.2. Additional Conditions

In this subsection, we suggest some conditions which say that particular transactions must include at least certain other transactions in their prefix subsequences. The conditions presented here are meant to be examples only, and are not necessarily intended to hold for all SHARD-like systems and all transactions. These restrictions are useful in guaranteeing certain properties of executions, as we demonstrate in Section 5. On the other hand, they reduce system availability. System and application designers must weigh the correctness gained by restricting the prefix subsequences against the reductions in availability.

First, we say that execution e is *transitive* provided that the following condition holds. Let T, T' and T'' be transactions (i.e. transaction instances) occurring in e. If T' is in the prefix subsequence of T and T'' is in the prefix subsequence of T', then T'' is in the prefix subsequence of T. Transitivity is a natural requirement, ensuring a basic sort of consistency among the prefixes seen by related transactions.

Example:

The execution in the previous example fails to be transitive, but for a trivial reason. Namely, the REQUEST(P101) and REQUEST(P102) transactions are assumed to execute with complete prefixes. Since the MOVE–UP which generates move–up(P101) sees the effects of REQUEST(P101), transitivity would imply that this MOVE–UP should also see a complete prefix, which is not what happens. However, note that REQUEST and CANCEL transactions have only trivial decision parts, so they would cause the same updates to be generated no matter what prefix they see. Therefore, we can modify the execution slightly, assigning each of REQUEST(P101) and REQUEST(P102) the prefix subsequence consisting of the first 198 transactions, without changing the updates generated. The resulting modified execution is transitive.

Another restriction which might be useful in some cases is to require that some particular transaction T must run with the complete prefix. This might be useful for very crucial transactions, say for an audit transaction in a high-finance banking system: it might be desirable for audits to see the effects of all the preceding deposit, withdrawal and transfer transactions. Although we have not done so in this paper, it should be possible to prove strong correctness results about transactions running with complete prefixes.

Requiring a complete prefix is very restrictive. There are some variants on this condition which are less restrictive but still lead to some very useful properties. For example, we might limit the number of previous transactions which are not visible to a particular transaction. Namely, transaction T is said to be *k-complete* in execution e provided that, in e, T sees the results of all but at most k of the preceding transactions. The k-completeness condition, for a particular k, does not seem to be a natural requirement to impose on an implementation, since in general, it seems difficult to guarantee a reliable value for k. (It might be possible to obtain an estimate of this value by considering known characteristics of the message system together with the expected rate of transaction processing.) However, k-completeness seems to be more useful as a hypothesis for conditional claims which describe the behavior of the system in different situations, for different values of k.

Another kind of condition which limits the amount of concurrency is as follows. Let G be a group of transaction instances. We say that group G is *centralized* in execution e provided that, in e, each of the transactions in G includes in its prefix subsequence all the others from G which precede it in the complete prefix. For example, it might be useful to centralize all the transactions which could cause the cost of a particular integrity constraint to become nonzero (e.g. all the withdrawal transactions, in a banking system). This strategy might be used to guarantee that this cost can never become nonzero. Alternatively, it might be useful to centralize all the transactions which affect a particular object, or a particular portion of the database. This strategy might be used to guarantee serializable execution for those objects or portions of the database.

If the system guarantees that transactions in G are centralized, it might be useful for the application programmers and users to imagine the existence of a centralized "agent" for G. For instance, it might be useful for users of the airline system to think of a single agent who manages all the MOVE−UPs and MOVE−DOWNs, i.e. all the movement between WAIT−LIST and ASSIGNED−LIST. This abstraction could be useful even if there is actually no such centralized agent, but rather if (using some locking strategy, for example), the agent is implemented in a distributed way.

Some specific groupings for the airline reservation system are discussed in detail in Section 5, along with examples of correctness conditions that result from this requirement.

The final condition presupposes a notion of time. A *timed execution* is an execution, together with a nonnegative real number ("real time") for each transaction instance. These real times are intended to model the times at which the transactions are initiated. In the event that the transaction order is determined by timestamps, these real times need not be the same as the timestamps, and in fact the real times need not even be ordered in the same way as the transaction sequence. However, if the order of real times is monotonic, we say that the timed execution is *orderly*. An execution is said to have *t-bounded delay* provided that the prefix subsequence of each transaction T includes every transaction in the prefix whose real time is at least t smaller than T's real time. Thus, each transaction can see the effect of every other transaction that precedes it in the transaction ordering and is not too recent.

3.3. Implementation Issues

It is very natural to use the conditions described in the preceding subsections as the correctness conditions for the distributed system described in the Introduction. The system is able to assign timestamps in some way so as to determine a total ordering of the transactions. The transactions are initially executed at one node, and then information about the transactions is sent to the other nodes. The nodes can undo and redo actions in order to ensure that as new updates are seen, each succeeding update has the effect that it would if executed in a complete prefix state. There are a number of optimizations which allow the system to avoid undoing large numbers of transactions [BK], and optimized storage structures make this process even more efficient [SKS].

The updates only are sent around, and are undone and redone to yield a sequential ordering. The fact that the decision parts are not redone means that the system does not satisfy the usual notion of serializability; however, the system does satisfy the prefix subsequence property, i.e. that every transaction sees the effects of a subsequence of its prefix.

It should be clear that an appropriate distributed communication protocol could guarantee transitivity, perhaps by piggybacking information about known transactions on messages.

There are a number of ways that a system could guarantee the subsequence restrictions described in the previous subsection. For instance, consider centralization of the transactions in G. It is possible to force all the transactions in G to run at the same node of a distributed system. Alternatively, a transaction in G with timestamp t might have to wait till it receives messages from all nodes saying "I will issue no more G transactions with timestamp earlier than t." This type of concurrency control might significantly reduce system availability. The probabilistic concurrency control methods of [S] provide other techniques for obtaining centralization.

4. Conditions Guaranteed by the Transactions

This section describes conditions which might be guaranteed by the transactions, analogous to preservation of integrity constraints in the usual development. We do not intend to require that all of these conditions hold for all sets of transactions; rather, we expect different conditions to be useful in different applications. We attempt to formulate the conditions in a general way, so that they might apply to different resource allocation applications. We describe how the conditions apply to the airline reservation system.

The first subsection defines some conditions involving costs of database states. Update parts of transactions are analyzed to determine whether or not they have the potential of increasing the cost, or are guaranteed to decrease the cost, with respect to a particular integrity constraint.

The second subsection discusses conditions involving fairness, a property particularly important in applications in which certain entities compete for access to some resource or service. We define priority among competing entities, and prove that certain conditions ensure that transactions preserve priority.

We define an *application* to consist of a collection of database states, (including designation of initial and well-formed states), their integrity constraint information (including costs), and a set of transactions. The properties we describe in this section are properties of applications.

4.1. Conditions Involving Costs

We say that an application is *initially zero cost* provided that $\text{Cost}(s_0) = 0$. That is, all the integrity constraints are satisfied in the initial database state. Clearly, the airline system is initially zero cost.

Another interesting property would be that a transaction T "preserves integrity", just as it is required to do in the usual concurrency control theory. A formal statement of this property might be: "If s is a well-formed state with cost(s) = 0, and if T(s,s) = s', then cost(s') = 0." This says that if T runs so that it changes the same state that it sees, then it does not cause a violation of the integrity constraints if they were previously satisfied. (We might say that T does not cause a violation of the integrity constraints "on purpose".) In the present setting, a more general kind of condition is appropriate, which also involves the behavior of transactions when the costs are nonzero.

We begin by describing a very strong property of a transaction T that says that there is no possibility of T ever causing an increase in the cost for constraint i. An update A is said to be *increasing* for constraint i provided that there is some well-formed s for which cost(A(s),i) > cost(s,i). That is, the update has the potential of increasing the cost of constraint i, although it need not actually do so in all circumstances. Otherwise, i.e. if the update could never increase the cost of constraint i, A is said to be *non-increasing* for constraint i. A transaction T is *safe* for constraint i provided that the following holds. If s is a well-formed state and $D_T(s) = A$, then A is nonincreasing for constraint i. Otherwise, i.e. if there is some well-formed s for which $D_T(s)$ is increasing, then we say that T is *unsafe* for constraint i.

Example:

In the airline system, the request(P) update is nonincreasing for the overbooking constraint, but is increasing for the underbooking constraint, since in states with fewer than 100 assigned people, and with P not already waitlisted or assigned, this request causes an increase in cost (of \$300). The cancel(P) update is also nonincreasing for the overbooking constraint, but is increasing for the underbooking constraint, since in states with at most 100 assigned people (including P) and sufficiently many waitlisted people, this cancellation causes an increase in cost (of \$300). On the other hand, the move--up(P) update is increasing for the overbooking constraint, since in states with at least 100 assigned people, this move-up causes an increase in cost (of $900). However, it is nonincreasing for the underbooking constraint. Finally, the move−down(P) update is nonincreasing for the overbooking constraint, but is increasing for the underbooking constraint since in states with at most 100 assigned people, this move-down causes an increase in cost (of $300).

Example:

The only updates that are increasing for the overbooking constraint are those of the form move−up(P). Since only the MOVE−UP transaction can generate a move−up(P) update, the other transactions are all safe for the overbooking constraint. However, the MOVE−UP transaction is unsafe for the overbooking constraint. On the other hand, the MOVE−UP transaction is safe for the underbooking constraint, but the other three transactions are all unsafe for the underbooking constraint.

A less restrictive, interesting property to consider might be intuitively described as: "Transaction T does not increase the cost of integrity constraint i on purpose." One simple formal way of stating this property is: "If s is a well-formed state and if T(s,s) = s', then cost(s',i) $\leq$ cost(s,i)." For technical reasons, we define a slightly stronger formulation, as follows.

We say that transaction T *preserves the cost* of constraint i provided that the following holds. If s is a well-formed state, T(s,s) = s', $D_T(s) = A$ and A is increasing for constraint i, then cost(s',i) = 0. That is, the decision part of a transaction T will only invoke an update part that (potentially) increases the cost of constraint i, when the state that T believes will exist after the update runs, will have a cost of 0 for constraint i. It is easy to see that this condition implies the simpler formulation described above. Also, it is obvious that if T is safe for constraint i, then it preserves constraint i.

Example:

We show that all transactions preserve the cost of the overbooking constraint. Since all transactions except for the MOVE−UP transaction are safe for the overbooking constraint, they preserve the overbooking constraint. The MOVE−UP transaction is unsafe for the overbooking constraint, so more argument is required in this case. The MOVE−UP transaction only generates a move−up(P) update from a state s for which AL(s) < 100 and WL(s) > 0. Then the state s' resulting from applying the move−up(P) update to s has AL(s') $\leq$ 100, and thus cost(s',1) = 0.

Now consider the underbooking constraint. The MOVE−UP transaction is safe for the underbooking constraint, and hence preserves the cost of the underbooking constraint. We also show that the MOVE−DOWN transaction preserves the cost of the underbooking constraint. The MOVE−DOWN transaction only generates an update which is increasing for the underbooking constraint from a state s for which AL(s) > 100. Then the state s' resulting from applying the update to s has AL(s') $\geq$ 100, and thus cost(s',2) = 0.

On the other hand, it is easy to see that REQUEST(P) and CANCEL(P) transactions do not preserve the cost of the underbooking constraint.

Since we are working in a setting in which integrity constraints are not always satisfied. i.e. costs may be nonzero, another useful property of transactions might be that they actually reduce the cost, not just preserve it. A transaction which reduces the cost for an integrity constraint can be regarded as a "compensating transaction" for violations of that integrity constraint. One possible formulation is as follows. We say that transaction T *compensates* for constraint i provided that the following holds. If s is well-formed, T(s,s) = s', and cost(s,i) > 0, then cost(s',i) < cost(s,i).

Lemma 1: Assume that all costs are integral. Assume that T compensates for constraint i. Then for any well-formed s, either cost(s,i) = 0, or there is some integer k > 0 such that $T(s,s) = s_1$, $T(s_1,s_1) = s_2,\dots,T(s_{k-1},s_{k-1}) = s_k$ and $\text{cost}(s_k,i) = 0$.

Proof: By repeated application of the definition. ∎

This lemma implies that if compensating transactions are run atomically from any point in an execution, using any available prefix subsequence, they will eventually result in an apparent state in which the cost of the constraint is 0. This idea can be stated formally as follows.

Corollary 2: Assume that all costs are integral. Assume that T compensates for constraint i. Let e be any finite execution, $\mathcal{U}$ any subsequence of the indices of e, and t the result of the updates indexed by $\mathcal{U}$, applied to s_0.

Then either cost(t,i) = 0, or else there is an extension of e to another execution, by an atomic suffix consisting of T's only, such that the prefix subsequence of the first T in the suffix is $\mathcal{U}$, t' is the apparent state after the last transaction, and cost(t',i) = 0.

Example:

It is easy to see that the MOVE–UP transaction compensates for the underbooking constraint, and the MOVE–DOWN transaction compensates for the overbooking constraint. In fact, it is possible to show that from any well-formed state, any atomic sequence of intermingled MOVE–UP and MOVE–DOWN transactions which contain sufficiently many of each will eventually reach an apparent cost of 0 for both integrity constraints.

Our last property involving costs, bounds the increase in cost that can result from the execution of a bounded number of transactions. First, we say that $s \leq_k t$ provided that there is a sequence of updates leading from s_0 to s, and a subsequence of that sequence containing all but at most k of the updates, such that the result of the subsequence applied to s_0 is t. That is, state t contains all the information in state s, except possibly for the effects of at most k updates. Then we say that function f *bounds the cost increase* for integrity constraint i provided that the following holds. For well-formed states s and t, if $s \leq_k t$, then $\text{cost}(s,i) \leq \text{cost}(t,i) + f(k)$. Thus, f(k) bounds the increase in the cost of integrity constraint i that can be incurred by k transactions.

Example:

In the airline reservation system, it is easy to see that 900k bounds the cost increase for the overbooking constraint, while 300k bounds the cost increase for the underbooking constraint.

Lemma 3: Let $\mathcal{U}$ be an atomic subsequence in execution e. Let s be the actual state before $\mathcal{U}$, and s' the actual state after $\mathcal{U}$. Let t be the apparent state before $\mathcal{U}$, and t' the apparent state after $\mathcal{U}$. If $s \leq_k t$, then $s' \leq_k t'$.

Proof: Straightforward. ▮

4.2. Conditions Involving Fairness

Another property of interest in some applications, i.e. those in which certain entities compete for access to some resource or service, is "fairness". In order to be able to state fairness conditions, we extend our application model to include the competing entities. In each state, we designate certain entities as "known" (i.e. currently competing). Also, in each state, we assume that there is a partial order on the known entities which describes priority.

We say that transaction T *preserves priority* provided that the following condition holds. If s is a well-formed state and T(s,s) = s', then: (a) If P and Q are both known in s and also in s', and if P precedes Q in s, then P precedes Q in s'. (b) If P is known in s and Q is not, and P and Q are both known in s', then P precedes Q in s'.

Example:

In our example, the people are the competing entities. In any state s, the known people are those on the WAIT−LIST or the ASSIGNED−LIST, in s. For P and Q known in s, we define P < Q to mean that either P precedes Q on the WAIT−LIST, or P precedes Q on the ASSIGNED−LIST, or else P is on the ASSIGNED−LIST and Q is on the WAIT−LIST. Then all of the transactions preserve priority.

A stronger property is also of interest. We say that transaction T *strongly preserves priority* provided that the following condition holds. If s and s' are well-formed states and T(s,s') = s'', then: (a) If P and Q are both known in s' and also in s'', and if P precedes Q in s', then P precedes Q in s''. (b) If P is known in s' and Q is not, and P and Q are both known in s'', then P precedes Q in s''.

Example:

It is easy to see that the REQUEST and CANCEL transactions strongly preserve priority, but the MOVE−UP or MOVE−DOWN transactions do not. For example, consider the MOVE−UP transaction. Assume that in state s, person P is first on the WAIT−LIST, and that transaction T, run from state s, generates a move−up(P) update. In state s', P is on the

WAIT–LIST but is not the first person; person Q is first. Then the move-up(P) action still moves P to the end of the ASSIGNED–LIST, in this case moving it ahead of Q. We have $P > Q$ in state s', but $P < Q$ in state s''. Thus, the MOVE–UP transaction is capable of changing the relative priorities of P and Q.

Similar remarks hold for the MOVE–DOWN transaction.

5. Properties of the Airline Reservation System

This section illustrates how the ideas presented in the previous sections can be used to prove interesting properties of executions of a particular application, the Fly-by-Night Airline System. Where it is possible, we state the results in a general way, so that they might later be applied to other examples.

Proving properties of executions of SHARD-like systems is far more difficult than for systems that preserve serializability. It is necessary to consider how a transaction's updates will execute on arbitrary well-formed database states, not just the database state seen by the decision part. With current techniques, it is not easy to understand how transactions and updates will behave in all possible situations, just by examining the transaction code. Even some of the relatively simple-sounding results in this section have proofs that are somewhat delicate. Our hope is that more experience with examples and proofs of this sort will eventually make the task easier.

The first subsection gives a brief discussion of some policy decisions affecting priority, that were embodied in the application design. The second subsection proves upper bounds on the costs of database states that could result from running the airline reservation system. All the bounds in this subsection are proved using the assumption that transactions see the effects of all but at most k of the preceding transactions. The cost bounds are stated in terms of this k. The third subsection refines the necessary conditions for obtaining these cost bounds and sharpens the bounds. The results in this subsection require only that transactions see the results of certain critical preceding transactions, rather than arbitrary transactions.

The fourth subsection proves results which rely on "centralization" assumptions, i.e. that some transactions see all of the preceding transactions of a certain type. Using centralization, we prove that some integrity constraints can never be violated. The final subsection proves some fairness properties.

5.1. Policy Decisions

Transactions in every application embody certain policy decisions. This subsection contains two examples which illustrate the policy decisions embodied in the Fly-by-Night System.

Example:

Suppose that two REQUEST(P) transactions occur without an intervening CANCEL(P). Both

REQUEST(P) transactions generate request(P) updates. At some point, it might be necessary to determine the effect of a sequence of updates including both of these request(P) updates. Then the second request(P) would be applied to a state s which reflects the previous occurrence of the earlier request(P). Thus, P might be in WAIT−LIST(s) or ASSIGNED−LIST(s); in this case, the update is defined to have no effect. The policy embodied in this definition is that if a person P is already on the WAIT−LIST or ASSIGNED−LIST, and makes a duplicate request, the new request does not change P's original priority. Alternative policy decisions might cause the second request to alter the priority somehow.

Example:

It is possible for two MOVE−UP transactions to occur which invoke move−up(P) updates for the same P, without an intervening CANCEL(P), or MOVE−DOWN which invokes a move−down(P) update. This could happen if the second MOVE−UP transaction is initiated without the first in its prefix subsequence. At some point, it might be necessary to determine the effect of a sequence of updates including both of these move−up(P) updates. Then the second move−up(P) would be applied to a state s which reflects the previous occurrence of the earlier request(P). Then P could be in ASSIGNED−LIST(s); in this case, the update has no effect. The policy embodied in this definition is that if a person P is already on the ASSIGNED−LIST, a new attempt to assign him a seat does not alter P's previous priority. Alternative policy decisions might cause the second move−up(P) to alter the priority.

5.2. Cost Bounds Resulting from k-Completeness

In this subsection, we prove upper bounds on the costs of the states reachable by running the airline system. All the bounds in this subsection are proved using the k-completeness assumption, i.e. the assumption that transactions see the effects of all but at most k of the preceding transactions. We begin with some preliminary lemmas.

Lemma 4: Let e be an execution, and T a k-complete transaction instance in e. Let s be the actual state before T and s' the actual state after T, in e. Let t be the apparent state before T and t' the apparent state after T.

1. Then $s \leq_k t$ and $s' \leq_k t'$.

2. Let i be a constraint, and assume that f bounds the cost of constraint i. Then $\mathrm{cost}(s,i) \leq \mathrm{cost}(t,i) + f(k)$ and $\mathrm{cost}(s',i) \leq \mathrm{cost}(t',i) + f(k)$.

Proof: Straightforward. ∎

The following theorem shows that k-complete transactions that preserve the cost of a constraint are guaranteed not to make the cost of that constraint larger, (except in the special case that the cost is very small).

Theorem 5: Let e be an execution, and T a k-complete transaction instance in e. Let i be a constraint, and assume that f bounds the cost for constraint i. Assume that T preserves the cost of constraint i. Let s be the actual state before T and s' the actual state after T, in e. Then either $\mathrm{cost}(s',i) \leq \mathrm{cost}(s,i)$ or else $\mathrm{cost}(s',i) \leq f(k)$.

Proof: Let t be the apparent state before T and t' the apparent state after T. Then $t' = T(t,t)$. Assume that T invokes action A in execution e, i.e. that $D_T(t) = A$.

Assume that cost(s',i) > cost(s,i). Then A is increasing for constraint i. Since T preserves the cost of constraint i, it follows that cost(t',i) = 0. By Lemma 4, cost(s',i) ≤ cost(t',i) + f(k) = f(k). ▮

We can specialize the preceding results to obtain bounds for the airline system.

Corollary 6: Let e be an execution of the airline system, and T a k-complete transaction instance in e. Let s be the actual state before T and s' the actual state after T, in e.

1. If T is any transaction, then either cost(s',1) ≤ cost(s,1) or else cost(s',1) ≤ 900k.
2. If T is a MOVE−UP or MOVE−DOWN transaction, then either cost(s',2) ≤ cost(s,2) or else cost(s',2) ≤ 300k.

Proof:

1. By Lemma 5, the fact that all transactions preserve the overbooking constraint, and the fact that 900k bounds the cost increase for the overbooking constraint.
2. By Lemma 5, the fact that MOVE−UP and MOVE−DOWN transactions preserve the underbooking constraint, and the fact that 300k bounds the cost increase for the underbooking constraint.

▮

The previous results are enough to yield an upper bound for the overbooking cost (although not for the underbooking cost) in all reachable states. We obtain such an upper bound for the overbooking cost as a special case of the following more general theorem.

Theorem 7: Assume that the application has the property that all transactions preserve the cost of constraint i. Let e be an execution. Let f bound the cost of constraint i. Assume that all occurrences of transactions that are unsafe for constraint i, in e, are k-complete. Let s be any state reachable in e. Then cost(s,i) ≤ f(k).

Proof: The proof is by induction on the length of e. The basis, length 0, is immediate. For the inductive step, assume that the length of e is at least 1, and that T is the last transaction in e. Let s be the actual state before T, and s' the actual state after T.

The inductive assumption implies that cost(s,i) ≤ f(k). If cost(s',i) ≤ cost(s,i), the claim is immediate. So assume that cost(s',i) > cost(s,i); then T is unsafe for constraint i, and so T is k-complete in e, by assumption. Then Theorem 5 implies that cost(s',i) ≤ f(k), as needed. ▮

Our invariant upper bound on the overbooking cost follows as a corollary.

Corollary 8: Let e be an execution of the airline system. Assume that all MOVE−UP transactions are k-complete in e. Let s be any state reachable in e. Then cost(s,1) ≤ 900k.

Proof: By Theorem GENERAL−INVARIANT−BOUND, the fact that all transactions preserve the overbooking constraint, the fact that 900k bounds the cost increase for the overbooking constraint, and the fact that only MOVE−UP transactions are unsafe for the overbooking constraint. ▮

We would also like to obtain an analogous invariant upper bound for the underbooking cost.

Unfortunately, such a bound does not hold for our airline system, since it can fail in an execution where many requests or cancellations arrive in rapid succession without sufficient intervening MOVE−UPs. In order to prove an upper bound on the underbooking cost, it appears to be necessary to assume something about the MOVE−UP transactions occurring sufficiently frequently.

To be specific, we define a partition $\mathcal{G}$ of the indices of e into groups consisting of consecutive indices to be a *grouping* of e for constraint i provided that each group satisfies one of the following.
(a) It consists of exactly one index j, and transaction T_j preserves constraint i.
(b) If t is the apparent state after the group, then cost(t,i) = 0.
That is, we will consider instances of transactions that preserve the cost of constraint i individually, but we will consider other transactions together, paying special attention to points during the execution where the transactions believe they have reduced the cost of the constraint to 0. Of course, not every execution will have such a grouping, but if the application contains a compensating transaction for constraint i, Lemma 2 implies that executions with such groupings are abundant. The *normal* states of e, with respect to a particular grouping, are just those states which are reachable after the groups, i.e. the actual states after the groups.

The next theorem says that, if we restrict attention to normal states only, an invariant upper bound holds for the underbooking constraint.

Theorem 9: Let e be an execution and $\mathcal{G}$ a grouping of e for constraint i. Assume that f bounds the cost of constraint i. Assume that all transactions that preserve the cost of i, as well as all transactions that occur at the ends of groups, are k-complete in e. Let s be any normal state reachable in e. Then cost(s,i) $\leq$ f(k).

Proof: By induction on the length of e. The basis, length 0, is immediate. For the inductive step, assume that the length of e is at least 1, and that T is the last transaction in e. Let s be the actual state before T, and s' the actual state after T. Let t be the apparent state before T, and t' the apparent state after T. There are only two cases that need to be considered.

If T is the last transaction in a group, then cost(t',i) = 0. Since T is k-complete, Lemma 4 implies that cost(s',i) $\leq$ cost(t',i) + f(k), = f(k), as needed.

Otherwise, T is a transaction that preserves the cost of constraint i, and occurs alone in a group. Then s is a normal state in e. The inductive assumption implies that cost(s,i) $\leq$ f(k). If cost(s',i) $\leq$ cost(s,i), the claim is immediate. So assume that cost(s',i) > cost(s,i). Then Theorem 5 implies that cost(s',i) $\leq$ f(k), as needed. ∎

The preceding theorem specializes immediately to our example. The REQUEST and CANCEL transactions are the ones that do not preserve the underbooking constraint, while the MOVE−UP transaction compensates for that constraint. Thus, executions which have groupings for the underbooking constraint can be constructed by including a sequence of MOVE−UP transactions immediately after each REQUEST and after each CANCEL transaction.

Corollary 10: Let e be an execution and $\mathcal{G}$ a grouping of e for the underbooking constraint. Assume that all MOVE−UP and MOVE−DOWN transactions, as well as all transactions that occur at the ends of groups, are k-complete in e. Let s be any normal state reachable in e. Then cost(s,2) $\leq$ 300k.

Thus, under suitable k-completeness assumptions, combined with assumptions about frequency of compensating transactions, we can prove invariant upper bounds on the costs in all reachable states (or all normal reachable states).

The ideas used to prove the preceding results can be used to say more. Consider an execution e in which costs become very large (because k-completeness or frequency assumptions fail). If there is ever a time during the execution after which good completeness and frequency properties begin to hold, it is easy to see that correspondingly good upper bounds will be reestablished. For instance, we can get a result of this type for the underbooking constraint, using the ideas of Corollary AIRLINE−BOUND−4. If we assume that the required transactions are k-complete from some point on in the execution, then (once the next compensating group has occurred), the underbooking cost satisfies an upper bound of 300k. On the other hand, if we want to obtain a similar result for the overbooking cost, we cannot base it on the simple ideas of Corollary 8. Rather, we would have to use ideas similar to those used for the underbooking cost. At some point after k-completeness begins to hold in the execution, we would hypothesize a group of MOVE−DOWNs, bringing the apparent overbooking cost to 0, in order to compensate for any excess overbooking cost. With such a hypothesis, an eventual 900k bound on the overbooking cost could be proved. We omit formal statements of these results here.

It is possible to combine the results of Corollaries 8 and AIRLINE−BOUND−4 to get a single invariant upper bound on the total cost for the airline system. For example, we obtain the following.

Corollary 11: Let e be an execution and $\mathcal{G}$ a grouping of e for the underbooking constraint. Assume that all MOVE−UP and MOVE−DOWN transactions, as well as all transactions that occur at the ends of groups, are k-complete in e. Let s be any normal state reachable in e. Then cost(s) $\leq$ 900k.

Proof: Immediate from Corollaries 8, AIRLINE−BOUND−4 and the fact that every well-formed state has either cost(s,1) = 0 or cost(s,2) = 0. ∎

We finish this subsection with a closer look at the kinds of improvements that are guaranteed by compensating transactions. For example, it would be nice to have a lemma which says that a k-complete transaction which compensates for constraint i, is guaranteed to actually improve the cost of constraint i, unless that cost is small. Unfortunately, this is not true. Although the compensating transaction might "try" to improve matters, it is possible that, because of missing information from its own prefix, it might not succeed in doing so. For example, a MOVE−DOWN transaction might observe too many people on the

ASSIGNED−LIST, and might therefore invoke a move−down update. But if it happens to invoke a move−down for a person who had actually cancelled in the interim, that move−down will not improve the actual cost.

We do know, however, that running the transaction several times in succession (atomically) can guarantee actual improvement. More precisely, we obtain the following.

Lemma 12: Assume that all costs are integral. Let f bound the cost of constraint i. Assume that T compensates for constraint i. Let e be any finite execution, $\mathcal{U}$ any subsequence of the indices of e, containing all but at most k of the indices in e, and let s be the actual state after e.

Then either cost(s,i) $\leq$ f(k), or else there is an extension of e to another execution, by an atomic suffix consisting of T's only, such that the prefix subsequence of the first T in the suffix is $\mathcal{U}$, s' is the actual state after the last transaction, and cost(s',i) $\leq$ f(k).

Proof: Let t be the result of $\mathcal{U}$ applied to s_0. Then s $\leq_k$ t. By Corollary 2, either cost(t,i) = 0, or else there is an extension of e to another execution, by an atomic suffix consisting of T's only, such that the prefix subsequence of the first T in the suffix is $\mathcal{U}$, t' is the apparent state after the last transaction, and cost(t',i) = 0. If cost(t,i) = 0, then since s $\leq_k$ t, it follows that cost(s,i) $\leq$ cost(t,i) + f(k) = f(k), as needed. Otherwise, Lemma 3 implies that s' $\leq_k$ t', and so cost(s',i) $\leq$ cost(t',i) + f(k) = f(k), as needed. ▮

This theorem specializes to the airline system as follows.

Corollary 13: Let e be any finite execution of the airline system, $\mathcal{U}$ any subsequence of the indices of e, containing all but at most k of the indices in e, and let s be the actual state after e.

1. Either cost(s,1) $\leq$ 900k, or else there is an extension of e to another execution, by an atomic suffix consisting of MOVE−DOWNs only, such that the prefix subsequence of the first T in the suffix is $\mathcal{U}$, s' is the actual state after the last transaction, and cost(s',1) $\leq$ 900k.

2. Either cost(s,2) $\leq$ 300k, or else there is an extension of e to another execution, by an atomic suffix consisting of MOVE−UPs only, such that the prefix subsequence of the first T in the suffix is $\mathcal{U}$, s' is the actual state after the last transaction, and cost(s',2) $\leq$ 300k.

Thus, the cost bounds of this subsection limit the damage that can be caused when transactions operate with a bounded amount of missing information. As noted before, the bounds we obtain are intuitive rather than surprising. However, we know of no way to prove these sorts of intuitive statements in earlier frameworks.

We note that it is possible to obtain more refined versions of the results in this subsection. Generally, it is not actually necessary that the indicated transactions see all but k of the entire set of preceding transactions. Rather, only certain types of preceding transactions are important in each case, since they suffice to determine the results of critical decisions. For instance, in Corollary 8, it is not necessary that the MOVE−UPs be k-complete; for example, it would suffice for them to see all but k of the preceding MOVE−UP and REQUEST transactions. We examine this issue more closely in the next subsection.

5.3. More Refined Cost Bounds

In this subsection, we reconsider some of the results of the preceding subsection. We sharpen those results so that they only require that transactions see the results of certain critical preceding transactions, rather than arbitrary preceding transactions. The results in this subsection give detailed information that is specialized to our application; thus, they are not stated in very general terms. However, it seems that the general approach used in this subsection should extend to other applications.

We begin by proving some basic lemmas about sequences of updates. It is helpful to think of these results in terms of an automaton whose states represent (abstractions of) the global states of the database, and whose state-transitions represent the updates. (The decision parts of transactions are not modelled by this automaton.) The sequence of updates which occur in an execution is modelled by a path in the automaton. We are interested in identifying subsequences of a sequence of updates, which are guaranteed to lead to the same state in the automaton as does the whole sequence. If a transaction executes seeing only such a subsequence as its prefix subsequence, it would be guaranteed to have accurate information.

Let $\mathcal{A}$ be a sequence of updates (of the Fly-by-Night airline system) and P a person. As *assignment witness* for P in $\mathcal{A}$ is an ordered pair of updates, (A,B), from $\mathcal{A}$, satisfying the following conditions.
(a) A is a request(P) update, B is a move−up(P) update, and A precedes B in $\mathcal{A}$.
(b) There are no cancel(P) updates after A in $\mathcal{A}$.
(c) There are no move−down(P) updates after B in $\mathcal{A}$.

A *waiting witness* for P in $\mathcal{A}$ is either of the following:
(1) An update A, from $\mathcal{A}$, satisfying the following conditions.
   (a) A is a request(P) update.
   (b) There are no cancel(P) or move−up(P) updates after A in $\mathcal{A}$.
(2) A pair (A,B) of updates satisfying the following conditions.
   (a) A is a request(P) update, B is a move−down(P) update, and A precedes B in $\mathcal{A}$.
   (b) There are no cancel(P) updates after A in $\mathcal{A}$.
   (c) There are no move−up(P) updates after B in $\mathcal{A}$.

Recall that a person is *known* in a given state s if he is either in ASSIGNED−LIST(s) or WAIT−LIST(s).

Lemma 14: Let $\mathcal{A}$ be a sequence of updates, and s the state resulting from applying $\mathcal{A}$ to s_0. Let P be a person.
(a) P is known in state s exactly if there is a request(P) update in $\mathcal{A}$ which is not followed by a cancel(P) update.
(b) P is in ASSIGNED−LIST(s) exactly if there is an assignment witness for P in $\mathcal{A}$.
(c) P is in WAIT−LIST(s) exactly if there is a waiting witness for P in $\mathcal{A}$.

Proof: By analysis of the possible state transitions. ∎

For the next several lemmas, we use the following notation. Let $\mathcal{A}$ be a finite sequence of updates and let $\mathfrak{B}$ be a subsequence of $\mathcal{A}$. Let s be the state which results from applying $\mathcal{A}$ to s_0, and let t be the state which results from applying $\mathfrak{B}$ to s_0. The next lemmas relate the states s and t.

Lemma 15: Let P be a person. Assume that P is in ASSIGNED−LIST(s), and let (A,B) be an assignment witness for P in $\mathcal{A}$. Assume that $\mathfrak{B}$ contains both updates A and B. Then P is in ASSIGNED−LIST(t).

Proof: By definition of an assignment witness, A is a request(P) update, B is a move−up(P) update, and A precedes B in $\mathcal{A}$. Also, $\mathcal{A}$ contains no cancel(P) updates after A and no move−down(P) updates after B. Now, $\mathfrak{B}$ contains both A and B, in that order. Also, $\mathfrak{B}$ cannot contain any cancel(P) updates after A or move−down(P) updates after B, since there are none in $\mathcal{A}$. Thus, (A,B) is an assignment witness for P in $\mathfrak{B}$. Lemma 14 implies that P is in ASSIGNED−LIST(t). ∎

Lemma 16: Let P be a person. Assume that P is in WAIT−LIST(s). Assume that at least one of the following holds.
(a) A is a waiting witness for P in $\mathcal{A}$, and $\mathfrak{B}$ contains update A.
(b) (A,B) is a waiting witness for P in $\mathcal{A}$ and $\mathfrak{B}$ contains both updates A and B.
Then P is in WAIT−LIST(t).

Proof: Similar to the proof of Lemma TWO. ∎

The preceding two lemmas will be applied in cases where $\mathcal{A}$ denotes the entire sequence of updates preceding a particular transaction T, while $\mathfrak{B}$ denotes the subsequence of updates actually seen by T. The lemmas imply that if T sees certain of the preceding transactions, and a person P is actually on the ASSIGNED−LIST or WAIT−LIST, then T is guaranteed to know it. On the other hand, the next several lemmas deal with the opposite implication; they describe circumstances under which a transaction that believes that a person P is actually on the ASSIGNED−LIST or WAIT−LIST, is guaranteed to be correct.

Lemma 17: Let P be a person. Assume that $\mathfrak{B}$ contains the last cancel(P) update, if any, in $\mathcal{A}$. If P is known in t, then P is known in s.

Proof: Assume P is known in t. Then Lemma 14 implies that there is a request(P) update in $\mathfrak{B}$ which is not followed by a cancel(P) update in $\mathfrak{B}$. This request(P) update also occurs in $\mathcal{A}$, and there are no cancel(P) updates after the request(P) in $\mathcal{A}$, since $\mathfrak{B}$ contains the last cancel(P) update from $\mathcal{A}$. Therefore, Lemma 14 implies that P is known in s. ∎

Lemma 18: Let P be a person. Assume that $\mathfrak{B}$ contains the last move−down(P) update, if any, in $\mathcal{A}$. Also assume that $\mathfrak{B}$ contains the last cancel(P) update, if any, in $\mathcal{A}$. If P is in ASSIGNED−LIST(t), then P is in ASSIGNED−LIST(s).

Proof: Assume that P is in ASSIGNED−LIST(t). Then Lemma 14 implies that there is an assignment witness (A,B), for P in $\mathfrak{B}$. Thus, A is a request(P) update and B is a move−up(P) update, A precedes B in $\mathfrak{B}$, there are no cancel(P) updates in $\mathfrak{B}$ after A and there are no move−down(P) updates in $\mathfrak{B}$ after B. Updates A and B also appear in $\mathcal{A}$, in that order. There are no cancel(P) updates after A in $\mathcal{A}$, since $\mathfrak{B}$ contains the last cancel(P) update (if any) in $\mathcal{A}$. Similarly, there are no move−down(P) updates after B in $\mathcal{A}$. Thus, (A,B) is an assignment witness for P in $\mathcal{A}$. Lemma 14 implies that P is in ASSIGNED−LIST(s). ∎

Lemma 19: Let P be a person. Assume that $\mathfrak{B}$ contains the last move−up(P) update, if any, in $\mathcal{A}$. Also assume that $\mathfrak{B}$ contains the last cancel(P) update, if any, in $\mathcal{A}$. If P is in WAIT−LIST(t), then P is in WAIT−LIST(s).

Proof: Analogous to the proof of Lemma ONE. ∎

Again, we can apply the preceding three lemmas to the case where $\mathcal{A}$ denotes the entire sequence of updates preceding a particular transaction T, and $\mathfrak{B}$ denotes the sequence of updates actually seen by T. The lemmas imply that if T sees certain of the preceding transactions, then T is guaranteed to know that a particular P is not on the ASSIGNED−LIST or WAIT−LIST.

Now we can prove refined versions of the results of the previous subsection. Since the notation and details become somewhat unwieldy, we present versions of Corollaries 6 and 13 only, and omit the others.

Theorem 20: Let e be an execution of the airline system, and T a transaction instance in e. Let s be the actual state before T and s' the actual state after T, in e.

1. Assume that there are at most k persons P such that P is in ASSIGNED−LIST(s) but the prefix subsequence seen by T fails to include an assignment witness for P. Then either cost(s',1) $\leq$ cost(s,1) or else cost(s',1) $\leq$ 900k.

2. Assume that T is a MOVE−UP or MOVE−DOWN transaction. Assume that there are at most k persons P such that P is not in ASSIGNED−LIST(s) but the prefix subsequence seen by T fails to include either the last cancel(P) or the last move−down(P) from $\mathcal{A}$. Then either cost(s',2) $\leq$ cost(s,2) or else cost(s',2) $\leq$ 300k.

Proof: Let t be the apparent state before T and t' the apparent state after T. Then t' = T(t,t). Assume that T invokes action A in execution e, i.e. that $D_T(t) = A$.

1. Assume that cost(s',1) > cost(s,1). Then T is a MOVE−UP transaction, A is a move−up update, and AL(t) < 100. For all persons P in ASSIGNED−LIST(s), except for the k exceptions described in the hypothesis, Lemma 15 implies that P is in ASSIGNED−LIST(t). Therefore, AL(s) $\leq$ AL(t) + k < 100 + k. It follows that AL(s') $\leq$ 100 + k, and so cost(s',1) $\leq$ 900k.

2. Assume that cost(s',2) > cost(s,2). Then T is a MOVE−DOWN transaction, A is a move−down update, and AL(t) > 100. For all persons P in ASSIGNED−LIST(t), except for the k exceptions described in the hypothesis, Lemma 18 implies that P is in ASSIGNED−LIST(s). Therefore, AL(s) $\geq$ A(t) - k > 100 - k. It follows that AL(s') $\geq$ 100 - k, and so cost(s',2) $\leq$ 300k.

∎

Theorem 21: Let e be any finite execution of the airline system, $\mathfrak{U}$ any subsequence of the indices of e, and let s be the actual state after e.

1. Assume that there are at most k persons P such that P is in ASSIGNED−LIST(s) but $\mathfrak{U}$ fails to include an assignment witness for P.
   Then either cost(s,1) $\leq$ 900k, or else there is an extension of e to another execution, by an atomic suffix consisting of MOVE−DOWNs only, such that the prefix subsequence of the first T in the suffix is $\mathfrak{U}$, s' is the actual state after the last transaction, and cost(s',1) $\leq$ 900k.

2. Assume that there are at most k persons P such that P is in WAIT−LIST(s) but $\mathfrak{U}$ fails to include a waiting witness for P. Also assume that for all but at most k persons P, if P is not

in ASSIGNED−LIST(s), then $\mathfrak{U}$ includes the last cancel(P) (if any) from e, and $\mathfrak{U}$ includes the last move−down(P) (if any) from e.
Then either cost(s,2) $\leq$ 300k, or else there is an extension of e to another execution, by an atomic suffix consisting of MOVE−UPs only, such that the prefix subsequence of the first T in the suffix is $\mathfrak{U}$, s' is the actual state after the last transaction, and cost(s',2) $\leq$ 300k.

Proof: Let t be the result of $\mathfrak{U}$ applied to s_0.

1. By Corollary 2, either cost(t,1) = 0, or else there is an extension of e to another execution, by an atomic suffix consisting of MOVE−DOWNs only, such that the prefix subsequence of the first T in the suffix is $\mathfrak{U}$, such that t' is the apparent state after the suffix, and cost(t',1) = 0.

First assume cost(t,1) = 0. Then AL(t) $\leq$ 100. Let P be any person in ASSIGNED−LIST(s). If P is not one of the k exceptions described in the hypothesis, then Lemma 15 implies that P is in ASSIGNED−LIST(t). It follows that AL(s) $\leq$ AL(t) + k $\leq$ 100 + k, so cost(s,1) $\leq$ 900k, as needed.

Second, assume that the extension exists. Then AL(t') $\leq$ 100. Let the actual state after the suffix be s'. Let P be any person in ASSIGNED−LIST(s'). Then P is also in ASSIGNED−LIST(s), since the suffix does not add anyone to the assigned list. If P is not one of the k exceptions described in the hypothesis, then Lemma 15 implies that P is in ASSIGNED−LIST(t). None of the MOVE−DOWNs in the suffix could have generated a move−down(P), since if one did, then P would not be in ASSIGNED−LIST(s'). Therefore, P is in ASSIGNED−LIST(t'). It follows that AL(s') $\leq$ AL(t') + k $\leq$ 100 + k, so cost(s',1) $\leq$ 900k.

2. By Corollary 2, either cost(t,2) = 0, or else there is an extension of e to another execution, by an atomic suffix consisting of MOVE−UPs only, such that the prefix subsequence of the first T in the suffix is $\mathfrak{U}$, t' is the apparent state after the suffix, and cost(t',2) = 0.

First assume cost(t,2) = 0. Then either AL(t) $\geq$ 100 or else WL(t) = 0. Let P be any person in WAIT−LIST(s). If P is not one of the k exceptions described in the hypothesis, then Lemma 16 implies that P is in WAIT−LIST(t). It follows that WL(s) $\leq$ WL(t) + k. Let P be any person in ASSIGNED−LIST(t). If P is not one of the k exceptions described in the hypothesis, then Lemma 18 implies that P is in ASSIGNED−LIST(s). It follows that AL(t) $\leq$ AL(s) + k. Thus, either WL(s) $\leq$ k or else AL(s) $\geq$ 100 - k. Thus, cost(s,2) $\leq$ 300k.

Second, assume that the extension exists. Then either AL(t') $\geq$ 100 or else WL(t') = 0. Let the actual state after the suffix be s'. Let P be any person in WAIT−LIST(s'). Then P is also in WAIT−LIST(s), since the suffix does not add anyone to the wait list. If P is not one of the k exceptions described in the hypothesis, then Lemma 16 implies that P is in WAIT−LIST(t). None of the MOVE−UPs in the suffix could have generated a move−up(P), since if one did, then P would not be in WAIT−LIST(s'). Therefore, P is in WAIT−LIST(t'). So WL(s') $\leq$ WL(t') + k.

Now let P be any person in ASSIGNED−LIST(t'). Then P must be known in t, since otherwise the move−ups in the suffix could not put P into ASSIGNED−LIST(t'). If P is in ASSIGNED−LIST(t), and P is not one of the k exceptions described in the hypothesis, then Lemma 18 implies that P is in ASSIGNED−LIST(s) and hence in ASSIGNED−LIST(s'). On the other hand, if P is in WAIT−LIST(t), and P is not one of these same k exceptions, then

Lemma 17 implies that P is known in s. Since P is in ASSIGNED−LIST(t'), a move−up(P) occurs in the suffix. Then P is in ASSIGNED−LIST(s'). So AL(s') $\geq$ AL(t') - k. It follows that either WL(s') $\leq$ k or AL(s') $\geq$ 100 - k. In either case, cost(s',2) $\leq$ 300k.

▮

It is also possible to give refined versions of Corollaries 8, 10, and 11. We omit the details.

5.4. Cost Bounds Resulting from Centralization

In this subsection, we give two results which describe conditions under which overbooking cannot occur at all. These conditions involve fairly strong centralization assumptions. The basic idea is that if all the move−up decisions are made centrally, it should not be possible to overbook. However, in order to prove this result, it is necessary for us to make some technical restrictions involving the requests.

Theorem 22: Let e be a transitive execution. Assume that the MOVE−UP transactions are centralized in e. Assume that for each P, the transactions that generates updates involving P are centralized in e. Let s be any state reachable in e. Then cost(s,1) = 0.

Proof: The proof is by induction on the length of e. The base case, where the length of e is 0, is easy. So assume that the length of e is at least one. Let T be the last transaction in e. Let t be the apparent state before T and t' the apparent state after T. Let s be the actual state before T, and s' the actual state after T. Let $\mathcal{A}$ be the actual sequence of updates preceding T, and let $\mathfrak{B}$ be the sequence whose effects are seen by T.

The inductive assumption says that cost(s,1) = 0. The only way that cost(s',1) can be nonzero is if T is a MOVE−UP transaction which generates a move−up update. Then AL(t) < 100.

We claim that ASSIGNED−LIST(s) $\subseteq$ ASSIGNED−LIST(t). If this is so, then AL(s) < 100, so AL(s') $\leq$ 100 and cost(s',1) = 0, as needed.

So fix P in ASSIGNED−LIST(s). Then there is an assignment witness for P in $\mathcal{A}$. The move−up(P) of the pair also appears in $\mathfrak{B}$, since the MOVE−UP transactions are centralized. The request(P) of the pair appears in the prefix seen by the move−up(P), since the transactions generating P updates are centralized. Therefore, the request(P) also appears in $\mathfrak{B}$, by transitivity. Thus, $\mathfrak{B}$ contains the assignment witness, and Lemma 15 implies that P is in ASSIGNED−LIST(t). ▮

The second result of this subsection is just a minor variant of the first, with an alternative technical restriction on the requests.

Theorem 23: Let e be a transitive execution. Assume that the MOVE−UP transactions are centralized in e. Assume that for each P, there is at most one REQUEST(P) transaction in e. Let s be any state reachable in e. Then cost(s,1) = 0.

Proof: The proof is nearly identical to the preceding one. The only difference is in the argument that the request(P) is in the subsequence seen by the move−up(P). We know that some request(P) appears in the subsequence seen by the move−up(P) action, for otherwise that action would not have been invoked. Since there is only one such request(P), the claim holds. ▮

Of course, it would be better if we could prove the same result only assuming centralization of MOVE−UP transactions and transitivity, and not making any assumptions about the transactions generating updates for the same person. But this stronger statement is not true, as is shown by the following example.

Example:

Consider an execution which consists of a succession of blocks of 4 transactions each,

REQUEST(P1), CANCEL(P1), REQUEST(P1), MOVE−UP,
REQUEST(P2), CANCEL(P2), REQUEST(P2), MOVE−UP,...,
REQUEST(P101), CANCEL(P101), REQUEST(P101), MOVE−UP.

The successive MOVE−UP transactions produce updates move−up(P1),..., move−up(P101). This execution is possible if each of the first 100 MOVE−UP transactions sees the first request in the same block, but not the cancel or the second request. The last MOVE−UP sees all the previous MOVE−UP's and the requests that they see, plus the cancels. Then this last MOVE−UP will think that the earlier MOVE−UP's acted erroneously, and that there is really no one on the assigned list. It will therefore decide to move P101 up. The cost after this execution is nonzero.

Similar results to those in this section should be provable, at least in principle, for the underbooking cost. However, the centralization assumptions that appear to be needed are so strong that the results do not seem very interesting.

5.5. Fairness

In this subsection, we consider fairness properties of the airline reservation system. As before, the results are stated in terms of the specific example, but the techniques appear to generalize to other applications.

For this section, we make the following very strong assumption. We assume that all MOVE−UP and MOVE−DOWN transactions are centralized; thus, there is essentially one "agent" making all decisions about seat assignment. It remains to be seen whether this assumption can be weakened, while still permitting proof of interesting fairness claims.

Recall the definition of passenger priority from Section 4.2: we say $P < Q$, for known P and Q, to mean that either P precedes Q on the WAIT−LIST, or P precedes Q on the ASSIGNED−LIST, or else P is on the ASSIGNED−LIST and Q is on the WAIT−LIST.

Lemma 24: Let $\mathcal{A}$ be a sequence of updates, and let $\mathfrak{B}$ be a subsequence of $\mathcal{A}$. Let P and Q be people. Assume that $\mathfrak{B}$ contains all move−up and move−down updates from $\mathcal{A}$. Also assume that $\mathfrak{B}$ contains all the request and cancel updates for P and Q, from $\mathcal{A}$. Let s be the result of $\mathcal{A}$ and t the result of $\mathfrak{B}$, applied to s_0. Then $P < Q$ in t if and only if $P < Q$ in s.

Proof: The updates in $\mathcal{A}$ which are not included in $\mathfrak{B}$ are only request and cancel updates for persons other than P and Q. These cannot affect the relative priority of P and Q. ∎

The following theorem says that, under certain restrictions, the relative priority of two requests is determined at the time the "agent" for MOVE−UP and MOVE−DOWN transactions first learns about both requests. Thus, except for an initial period of uncertainty during which the agent has not yet learned about the requests, their relative priority is fixed.

Theorem 25: Let e be a transitive execution. Assume that the MOVE−UP and MOVE−DOWN transactions are centralized. Let P and Q be people each of whom has exactly one REQUEST transaction, but no CANCEL transactions, in e. Let T be a MOVE−UP or MOVE−DOWN transaction having both REQUEST(P) and REQUEST(Q) in its prefix subsequence. Let t be the apparent state, and s the actual state, before T. If $P < Q$ in t, then also $P < Q$ in s and all other actual database states occuring later in e.

Proof: First, we show that $P < Q$ in s. Let $\mathcal{A}$ be the sequence of updates preceding T, and $\mathfrak{B}$ the subsequence actually seen by T. The centralization assumption implies that $\mathfrak{B}$ contains all move−up and move−down updates from $\mathcal{A}$. The other assumptions imply that $\mathfrak{B}$ contains all the request and cancel updates for P and Q, from $\mathcal{A}$. Then Lemma 24 implies that $P < Q$ in s.

Assume that T_1 is the first transaction (T or later) after which it is false that $P < Q$. Let t_1 be the apparent state before T_1 and t_1' the apparent state after T_1. Let s_1 be the actual state before T_1 and s_1' the actual state after T_1. Then $P < Q$ in s_1 but not in s_1'. The only possibility is that T_1 is a MOVE−UP or MOVE−DOWN transaction that causes the order of P and Q to become interchanged; thus, $Q < P$ in s_1'.

We claim that $P < Q$ in t_1. Let $\mathcal{A}$ be the sequence of updates preceding T_1, and let $\mathfrak{B}$ be the subsequence actually seen by T_1. $\mathfrak{B}$ contains all the moving updates from $\mathcal{A}$, by the centralization assumption. Also, $\mathfrak{B}$ contains the requests for P and Q, since the subsequence seen by T does, T is either equal to T_1 or else is in T_1's subsequence, and transitivity holds. Thus, applying Lemma 24, the orderings in t_1 and s_1 are the same, so $P < Q$ in t_1.

Now we claim that $Q < P$ in t_1'. This follows using Lemma 24, since $Q < P$ in s_1'. But if $P < Q$ in t_1 and $T_1(t_1,t_1) = t_1'$, then $P < Q$ in t_1', since all transactions preserve priority. This yields a contradiction. ∎

We can interpret the preceding theorem as follows. We might imagine that at the actual flight time, next January 1, the complete execution becomes known to the check-in attendant. The people that he actually allows to proceed onto the airplane are the 100 people who show up, who have the highest priority in the final database state. (CANCEL transactions can be run for the others, and then sufficiently many MOVE−UP or MOVE−DOWN transactions to cause AL to equal 100 or WL to equal 0.) If P and Q had previously become known to the "agent" for MOVE−UP and MOVE−DOWN transactions, with $P < Q$, and if P and Q both show up, if Q gets onto Flight 1, then so does P.

Example:

Our transaction definitions can lead to the following behavior for passengers' relative priorities. Assume that REQUEST(P) precedes REQUEST(Q), but the request(Q) update becomes known to the "agent" before the request(P) update. Then a move−up(Q) can occur, which moves Q up past P. Later, a move-down(Q) can occur. When this happens, our definitions say that Q gets put at the head of the WAIT-LIST, ahead of P. Subsequently, the moving agent can learn about the request(P) also. At that point, Q < P, so by Theorem 25, Q remains ahead of P. This happens even though there is sufficient information in the system to allow for Q to be placed on the WAIT-LIST after P, which is in keeping with their timestamp order for requests. Thus, the order obtained in the final state is determined by the order at the time a MOVE-UP or MOVE-DOWN transaction first sees both requests, but is not necessarily determined by the actual order in which the requests were initially made.

It is possible to redesign the application to respect the original request order in this situation. It suffices to include request timestamps explicitly in the database. Each of the two lists would always be kept sorted according to timestamp order. Thus, when the request(P) becomes known to the agent, he would insert P ahead of Q on the waiting list. (More precisely, when the move-down(Q) is run from a state in which P is on the waiting list, Q is not placed at the head of the waiting list, but rather is inserted in timestamp order, after P.) This relative position would be maintained from then on.

Theorem 25 makes a claim about relative priorities at times after a conceptual "agent" learns about two requests. In order for this condition to be meaningful as a correctness claim, the user must have a fairly detailed and sophisticated conceptual model of system operation, including prefix subsequences and agents. It might also be interesting to state fairness claims which involves a less detailed conceptual model. For example, we might want to state a condition which could be paraphrased aa follows. "If a REQUEST(P) is made sufficiently earlier than a REQUEST(Q), then P must precede Q in the final state." The following lemma can be used to infer such a property.

Lemma 26: Let e be a transitive execution. Assume that the MOVE-UP and MOVE-DOWN transactions are centralized. Let P and Q be people each of whom has exactly one REQUEST transaction, but no CANCEL transactions, in e. Assume that REQUEST(P) precedes REQUEST(Q) in e. Further assume that any MOVE-UP or MOVE-DOWN transaction that has REQUEST(Q) in its prefix also has REQUEST(P) in its prefix. Then P < Q in any actual state reached during e in which both P and Q are known.

Proof: Assume the contrary, and let T be the first transaction in e such that Q < P in the actual database state after T. Let t be the apparent state before and t' the apparent state after T. Let s be the actual state before and s' the actual state after T. Then Q < P in s' but not in s.

First, we claim that T must be a moving transaction. If T were a REQUEST(P) transaction, then the REQUEST(Q) cannot be reflected in s' since it occurs after REQUEST(P). All other cases can be ruled out by similar trivial arguments. So T is a moving transaction; thus, P and Q are known in s, so that P < Q in s. The only possibilities are that T is a MOVE-UP transaction that moves Q up past P, or that T is a MOVE-DOWN transaction that moves P down past Q. For either of these to happen, at least one of request(P) and request(Q) must be in the prefix subsequence of T.

Case 1: T has both request(P) and request(Q) in its prefix subsequence.
Then both P and Q are known in t. If P < Q in t, then Theorem 25 implies that P < Q in s', a contradiction. On the other hand, if Q < P in t, then Theorem 25 implies that Q < P in s, again a contradiction.

Case 2: T has only request(P), but not request(Q), in its prefix subsequence.
Then T must be a MOVE−DOWN which moves P down past Q. Therefore, Q must be in ASSIGNED−LIST(s). But in order for this to occur, there must be some MOVE−UP transaction T' appearing earlier than T in e, which moves Q up; clearly, request(Q) must be in the prefix subsequence of T'. T' is in the prefix subsequence of T, since the moving transactions are centralized. By transitivity, request(Q) is in the prefix subsequence of T. This is a contradiction. ∎

We can use this lemma to obtain a theorem of the form we described earlier, i.e. that if REQUEST(P) occurs sufficiently long before REQUEST(Q) (and other suitable conditions hold), then P retains priority over Q. All that is needed is an additional assumption that if REQUEST(P) occurs sufficiently long before REQUEST(Q), then any MOVE−UP or MOVE−DOWN transaction that has request(Q) in its prefix also has request(P) in its prefix.

Theorem 27: Let e be a transitive, orderly timed execution having t-bounded delay. Assume that the MOVE−UP and MOVE−DOWN transactions are centralized. Let P and Q be people each of whom has exactly one REQUEST transaction, but no CANCEL transactions, in e. Assume that REQUEST(P) precedes REQUEST(Q) by at least time t, in e. Then P < Q in any actual state reached during e in which both P and Q are known.

Proof: The t-bounded delay assumption and orderliness imply that any MOVE−UP or MOVE−DOWN that has REQUEST(Q) in its prefix also has REQUEST(P) in its prefix. The previous lemma then yields the result. ∎

6. Conclusions

In this paper, we have given precise correctness conditions for a highly available replicated database system such as CCA's SHARD. First, we gave basic definitions for the SHARD database and transaction model. We then described assumptions about how the system runs the transactions, followed by assumptions about applications. Finally, these two types of assumptions were combined to prove some interesting properties of a particular running application, an airline reservation system. Although the example is simple, it is illustrative of a large class of important resource-allocation problems.

The assumptions about how the system must run the transactions (in particular, the prefix subsequence condition) have been described in a very general way. They embody a new model for data processing, which is quite different from, and imposes new structure on, the traditional models used in concurrency control theory. We expect that this model will prove very fruitful for future research and for application design.

In describing our assumptions about the airline reservation application, we have tried to be as general as possible. The types of assumptions we have listed seem to be very appropriate for resource allocation applications, but we do not believe that they comprise a complete set of interesting application assumptions. It is likely that study of additional examples will yield other interesting types of assumptions as well.

The particular properties proved for our application involve bounds on the costs attributable to violations of integrity constraints, and fairness. For other resource allocation applications, similar cost bound and fairness results should be provable.

The system exhibits nonserializable behavior, so that being able to prove interesting conditions is an accomplishment. In the usual development, no guarantees at all can be proved in case information about any preceding transaction is missing. In contrast, we can prove interesting properties even with incomplete information. Moreover, small changes in available information lead to small changes in costs for integrity constraints.

The analysis required to obtain some of our results has been very delicate. This is because it is necessary to consider how updates will execute in many possible situations, not just from the database state seen by the decision parts of their transactions. Another difficulty is that SHARD does not impose any a priori restrictions on the kinds and orders of transactions that are submitted and processed. The need to consider the behavior of transactions in the presence of arbitrary preceding transactions, and arbitrary partial knowledge about the past, makes the analysis of SHARD transactions more difficult than for ordinary (serializable) transactions. But this kind of analysis seems unavoidable; whether or not a formal, mathematical analysis is carried out for a particular application, application programmers do need to consider, at least informally, how transactions will behave in the presence of arbitrary preceding transactions and arbitrary partial knowledge about the past. We provide a *framework* for this kind of analysis, but more needs to be done to develop appropriate styles of programming and *methods* of analysis.

A next step in this research should be the consideration of other example applications. Additional resource allocation examples should be examined, such as examples from banking and inventory control. Other, non-resource-allocation, examples should be studied. Some examples appropriate for SHARD might involve "distributed data structures". The highly-available distributed dictionary studied in [FM] is one example that fits the SHARD framework, and there should be others. Also, it has been claimed that name servers such as Grapevine [B] have interesting but nonserializable behavior; it seems likely that they can be described within our framework. Still other appropriate examples might arise from real-time control.

For each of these examples, simple prototypes could be defined, capturing the essential behavior of the example. Study of these prototypes should determine the appropriate properties to prove in each case. Cost bounds and fairness should reappear, but other properties should also be of interest. It is important to look for general methods of programming and analysis.

Other theoretical work also seems possible. For instance, we have described some interesting automaton structure in Section 5.3. This structure could be studied and generalized. Also, it should be possible to obtain complexity results. Particular examples of desirable application behavior could be studied individually, and costs (e.g. amount of communication, or local storage) determined for achieving correct behavior.

On the systems design side, SHARD itself needs to be generalized in at least two important ways. First, the inessential full replication assumption needs to be removed. Even with only partial replication, it should be possible to continue to maintain the correctness conditions we describe in this paper, by judicious assignment of data and transactions to nodes, (i.e. in such a way that each transaction will have copies of all the data it requires). It should even be possible to allow some of the data which transactions read to be present in summary form, rather than in its full detail. Second, the SHARD work needs to be integrated with earlier work on serializability. It should be possible to build an application system in which certain critical transactions run serializably, while the others run in a highly available manner. The application designer should be able to specify the modes of operation for different transactions. As the system design gets extended, the theory also needs to be extended to incorporate these two generalizations.

It is apparent to us that there is an interesting theory to be developed, for proving properties of nonserializable highly available replicated database systems. We believe that this paper gives some useful ideas on how to begin.

7. References

[AM] Allchin, J. E. and McKendry, M. S., "Synchronization and Recovery of Actions," *Proc. of the Second Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, Canada, August 17-19, 1983, pp. 31-44.

[B] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing," *Comm. of the ACM 25*, 4 (April 1982), pp. 260-274.

[BG] Bernstein, P. A., and Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* 13,2 (June 1981), pp. 185-221.

[BK] Blaustein, B. T. and Kaufman, C. W., "Updating Replicated Data During Communication Failures," *Proc. of the Eleventh Intl. Conf. on Very Large Databases*, Stockholm, Sweden, August 1985, pp. 49-58.

[FM] Fischer, M. J. and Michael, A., "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," *Proc. of the Symposium on Principles of Database Systems*, Los Angeles, California, March 29-31, 1982, pp. 70-75.

[G] Garcia-Molina, H., "Using Semantic Knowledge for Transaction Processing in a Distributed Database," Tech. Rep. 285, Princeton Univ. Dept. of Electrical Engineering and Computer Science, April 1981. Also appeared in *Transactions on Database Systems*, 8, 2 (June, 1983), pp. 186-213.

[GLBKSS] Garcia-Molina, H., Lynch, N. A., Blaustein, B. T., Kaufman, C. W., Sarin, S. K., and Shmueli, O., "Notes on a Reliable Broadcast Protocol," CCA technical report, 1985.

[J] Jefferson, D., "Virtual Time," *Transactions on Programming Languages and Systems*, (July 1985), 7, 3, pp. 404-425.

[S] Sarin, S. K., "Robust Application Design in Highly Available Distributed Databases", *Proc. Fifth Symp. Reliability in Distributed Software and Database Systems*, January 1986, pp. 87-94.

[SBK] Sarin, S. K., Blaustein, B. T., and Kaufman, C. W., "System Architecture for Partition-Tolerant Distributed Databases," *IEEE Transactions on Computers* C-34, 12 (December 1985), pp. 1158-1163.

[SKS] Sarin, S. K., Kaufman, C. W., and Somers, J. E., "Using History Information to Process Delayed Database Updates," CCA, 1986, submitted for publication.

[SL] Sarin, S. K., and Lynch, N. A., "Discarding Obsolete Information in a Replicated Database System," CCA, 1986, submitted for publication.