Implementing Fault-Tolerant Distributed Objects

Kenneth P. Birman
Thomas Joseph
Thomas Raeuchle
Amr El-Abbadi
TR 84-594
March 1984
Revised October 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

IMPLEMENTING FAULT-TOLERANT DISTRIBUTED OBJECTS

Kenneth P. Birman, Thomas A. Joseph, Thomas Raeuchle, Amr El Abbadi

Department of Computer Science
Cornell University, Ithaca, New York, 14853

ABSTRACT

This paper describes a technique for implementing k-resilient objects -- distributed objects that remain available and whose operations are guaranteed to progress to completion, despite up to k site failures. The implementation is derived from the object specification automatically, and does not require any information beyond what would be required for a non-resilient, non-distributed implementation. It is therefore unnecessary for an applications programmer to have knowledge of the complex protocols normally employed to implement fault-tolerant objects. Our technique is used in *ISIS*, a system being developed at Cornell to support resilient objects.

1. Introduction

Abstract data types [Lis74] have been proposed as a methodology for structuring both centralized and distributed software systems. An instance of an abstract data type, called an *object*, encapsulates some data, the *object-data*, and provides its users with a set of *operations* to manipulate these data. In object-based distributed systems, an operation is performed by invoking an object using a *remote procedure call* [Nel81], which passes value parameters to the object and returns the results of the operation to the caller. An operation may call another object; this gives rise to nested operations and nested remote procedure calls. The best guarantee that present distributed systems offer in the presence of site failures is that operations execute atomically: that they execute either completely or not at all, and if concurrent access to the object is permitted, that operations execute in a serializable manner. If a site at which an operation is being executed fails, the operation is either aborted or blocks until the site recovers. A nested transaction scheme [Mos81a, Mos81b] is used to implement this behavior. In this paper, we describe a technique to implement objects in such a way that their operations progress to completion even if up to a specified number of sites fail.

Checkpoints have been used in many systems to facilitate recovery from failures. Here we show how checkpoints can be used to obtain a *k-resilient* implementation of an object: one in which operations are guaranteed to progress to completion despite up to k site failures. We assume that processors satisfy a *fail-stop* assumption: that they fail by halting and do not produce spurious messages[1], and that active sites are notified of the failure of a site [Sch83]. The method we present does not require the object specification to contain any information beyond what is required for a non-resilient implementation. Thus a specification may be given in a language similar to MODULA [Wir77], ADA [Ich79], or ARGUS [Lis82]. A k-resilient implementation is derived from the specification automatically. The advantage of this approach is that a user wishing to develop a fault-tolerant system can do so easily, without having to program or even understand complex protocols for synchronization between sites. This technique is now being implemented in *ISIS*, a project at Cornell that aims to design and develop a system that provides support for the construction of resilient objects.

Fig. 1 depicts the specification of a simple object, a detailed description of which is given in Section 4. Note that there are two types of data: *object-data*, which remain in existence for the lifetime of the object, and *temporary data*, which are declared within an operation, and exist only while the operation is in execution. In this object, the variables *fee* and *seats_sold_here* are the object-data, while the variables *performance*, *seat*, *customer_acct*, *price*, and *final_price* declared within the operation *charge_seat* are examples of temporary data.

In this paper, we represent an execution of an operation O as a sequence of *actions* $\{a_1, a_2, \ldots, a_n\}$, where each a_i is a computation on the object-data, a computation on temporary data, or an operation on another object. In the last case, the action is called an *external action*. The *index* of an action a_i is i. The execution of a sequential operation, or each

[1] *Byzantine* or other malicious failures are not considered. An underlying failure detector can be implemented in software to obtain the desired failure behavior.

```
/* Vendor reserves theater tickets and charges cost to buyer's bank account      */
/*                                                                               */
/* Operations:                                                                   */
/*   show_seats     - displays seats available for a particular performance      */
/*   charge_seat    - reserves seat, transfers cost from buyer's bank account    */
/*                    to theater's account, transfers fee from buyer's account   */
/*                    to vendor's account. Returns TRUE if successful, FALSE     */
/*                    if seat is unavailable, or if balance is insufficient      */
/*   get_stat       - returns total number of seats sold by this vendor          */
/*   change_fee     - changes value of fee charged to customer                   */
/*   create_vendor  - creates new instance of vendor object                      */


3-resilient vendor
  created_by      create_vendor;          /* initializes new object instance */
  operations      show_seats, charge_seat, get_stat, change_fee;
  calls           bank, theater;
  constant        THEATER_ACCT, VENDOR_ACCT, UNAVAILABLE;

  integer         fee, seats_sold_here;


  show_seats (performance)
    integer performance;
    {
      theater.display (performance);          /* call display operation in */
                                              /* object theater            */
    }


  charge_seat (performance, seat, customer_acct)
    integer performance, seat, customer_acct;
    integer price, final_price;
    boolean enough;
    {
      price := theater.reserve (performance, seat);

      if price = UNAVAILABLE then
        return (FALSE);
      else
        final_price := price + fee;
        enough := theater.inquire (customer_acct, final_price);
        if enough then
          bank.transfer (customer_acct, THEATER_ACCT, price);
          bank.transfer (customer_acct, VENDOR_ACCT, fee);
          seats_sold_here := seats_sold_here + 1;
          return (TRUE);
        else
          theater.cancel (performance, seat);
          return (FALSE);
        endif;
      endif;
    }


  get_stat()
    {
      return (seats_sold_here);
    }
```

contd. . . .

```
change_fee (new_fee)
  integer new_fee;
  {
    fee := new_fee;
  }

create_vendor (init_fee)
  integer init_fee;
  {
    fee := init_fee;
    seats_sold_here := 0;
  }

end;
```

Figure 1: Specification of a resilient ticket vendor

branch computation of a parallel operation, is called an *activity*. Since objects support concurrent access, there may be several activities in an object at the same time, even if all its operations are sequential.

In Section 2 we describe how to implement a resilient object. Section 3 deals with optimizations, and Section 4 provides an example illustrating the technique of deriving a resilient implementation. In Section 5 we extend the technique to allow for total failures, and in Section 6 we discuss applications. Section 7 concludes the discussion.

2. Implementing a k-resilient object

K-resiliency imposes two constraints on an object implementation:

- *Availability* - Even after up to k sites fail, it must be possible to access (a copy of) the object-data.
- *Forward progress* - After up to k site failures, there must be enough information at the remaining active sites to continue the execution of any operation that was in progress.

To satisfy these constraints, a k-resilient object must be implemented at $k+1$ or more sites, because after k sites fail, there must be at least one operational site at which a copy of the object-data is available, and which is able to continue execution.

2.1. Availability

The availability constraint is addressed by replicating the object-data at all the sites at which an object resides and using a replicated data concurrency control algorithm to keep the copies consistent. Several such algorithms have been proposed in the literature, and Bernstein and Goodman have presented a theory for proving that they operate correctly in the presence of site failures [Ber83]. In most situations, *ISIS* uses a variant of the "available copies" algorithm proved correct in [Ber83].

2.2. Forward progress

The forward progress constraint is met using a coordinator-cohort scheme, in conjunction with checkpoints and retained results (Section 2.3). When an object is invoked to perform an operation $O = \{a_1, a_2, \ldots, a_n\}$, one of the sites at which the object resides is designated as the *coordinator*, and it supervises the activity. The other sites are called its *cohorts*. All the sites are equally capable of performing operations; hence any site could assume the role of coordinator. The most practical way to choose a coordinator is for the site that actually receives a message on behalf of an object to appoint itself as coordinator. It then informs the other sites that they are its cohorts for the execution of the operation requested in the message. An alternative is to run a consensus protocol among the sites, but this involves unnecessary communication. While executing O, the coordinator transmits enough information to its cohorts to ensure that O could be completed by any one of them if the coordinator fails at some point in the activity. This achieves k-resiliency because there are at least k cohorts - each time the coordinator fails, it is replaced by one of the remaining cohorts. Again, the new coordinator can be chosen arbitrarily; a consensus protocol is avoided if this choice is made on the basis of a predetermined ordering on the sites. If an action a_i is a call to another object, we assume that the same scheme is used in the called object to ensure that it is completed despite up to k' failures, where k' is the resiliency of the called object. We must have $k' \geq k$ for the calling object to be k-resilient.

2.3. Checkpoints and retained results

The information required for achieving forward progress is contained in *checkpoints* and *retained results*. The coordinator can establish a checkpoint before any action a_i by informing its cohorts of its local state. Action a_i is then the *location* of that

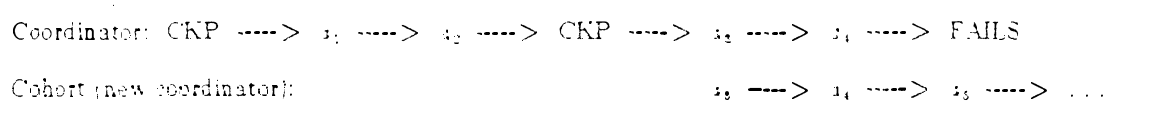

Figure 2.

checkpoint, which is said to *cover* actions a_1 through a_{i-1}. The data included in a checkpoint are the value of the instruction pointer, the values of all the object-data, and the values of the temporary variables; we will later show how the information transmitted at a checkpoint can be reduced considerably. If the coordinator fails, a cohort enters the state the coordinator had at the most recent checkpoint location and executes forward.

A separate mechanism is required if the failed coordinator could have performed external actions since the latest checkpoint. Consider Fig. 2. The coordinator establishes a checkpoint at the start of $O = \{a_1, a_2, \ldots, a_n\}$, one before a_3, and fails after executing a_4. The new coordinator uses the information in the latest checkpoint to set the values of the object-data and the temporary data to those they had before the failed coordinator executed a_3. It then resumes O from action a_3. If a_3 is an action on temporary or object-data, the new coordinator can simply re-execute it, because they were executed locally at the failed coordinator. If a_3 is an external action, however, the changes made to the called object are still in effect. Hence, the called object must not re-execute the operation when it is invoked by the new coordinator, because this could make the object-data of the called object inconsistent. However, the new coordinator needs to know the results of a_3 before it can execute a_4, because a_4 and subsequent actions could depend on the results of a_3.

Each activity is hence given a unique identifier, its *activity-ID*. The coordinator generates the activity-ID at the start of an activity, and makes it known to its cohorts before it executes the first action. Each action of an activity is then identified uniquely by its *action-ID*, which is formed from the activity-ID and the index of the action. When the coordinator performs an external action, it passes the action-ID to the invoked object. Every called object stores a copy of the results it returns in association with the action-ID. These are called *retained results*. If a cohort that takes over after a coordinator failure issues the same operation again, as in the example above, it uses the same action-ID. The called object thus recognizes this operation as having been executed already and does not re-execute it, but merely returns a copy of the retained results to the caller[2]. The caller proceeds as if the operation had been actually executed. To ensure that retained results remain available, they are transmitted to all the sites at which an invoked object resides before they are returned to the caller. Retained results can be viewed as a checkpoint taken in the called object when an execution terminates. Since the execution has terminated, there is no need to preserve the values of local data - only the final results need be saved.

A cohort can resume an activity from a checkpoint and execute forward if the results of all external actions executed after the latest checkpoint are retained. It follows that the forward progress constraint is satisfied if the coordinator establishes checkpoints in such a way that this is always true.

2.4. How long are results retained?

Since retained results take up storage space, it is desirable to save as few retained results, and for as short a time, as possible. An object X must retain the results of at least the last operation it performed for each activity that has invoked it, because in any distributed system, it is impossible to both take an action at a site and inform other sites about it as an atomic action, unless multi-phase protocols are carried out for every action. Consider Fig. 3, which depicts the progress of a coordinator in an object Y. Even though the coordinator establishes a checkpoint before every action, the checkpoint information is not sufficient for the cohorts to determine whether a_2 was executed or not, since they cannot distinguish between the two cases (a) and (b) of coordinator failure. The new coordinator starts by issuing a_2. If a_2 is an external action on X, and if X does not retain its results, a_2 would be executed twice in case (b), possibly leaving X inconsistent.

On the other hand, X need not retain the results of an operation that is covered by a checkpoint in the invoking activity, because it will never be re-issued by a cohort. One alternative is for the coordinator of an activity to instruct all previously called objects to discard their retained results each time it establishes a

(a) CKP ---> a_1 ---> CKP ---> FAILS

(b) CKP ---> a_1 ---> CKP ---> a_2 ---> FAILS

Figure 3.

[2] If the called object is an external device like a money-dispenser, a combination of software and hardware will be required to achieve this behavior.

new checkpoint. This leads to a large number of messages being sent, and is justifiable only if storage cost is extremely high. A more practical solution is for a called object to retain the results of only the last $n \geq 1$ operations it performs for each activity. The value of n is a property of an object implementation, and is used by a coordinator to decide when to establish checkpoints.

2.5. When are checkpoints established?

The coordinator establishes a checkpoint at the start of an activity to inform its cohorts of the activity-ID and to pass them the parameters of the operation. Subsequently, it establishes checkpoints in such a way that the results of all external actions not covered by a checkpoint are retained. To do this, the coordinator maintains a log of the actions it has executed so far and the location of the latest checkpoint. If it has executed n external actions on the same object since the last checkpoint, it establishes a new checkpoint before it invokes that object again[3]. Thus the forward progress constraint is satisfied.

2.6. Requirements of the concurrency control algorithm

A forward progress mechanism is meaningful only in the context of a concurrency control algorithm that does not require operations to be aborted and restarted in the event of a failure. Consider the case where a locking mechanism is used, with information about read locks being maintained only locally. If a coordinator fails after acquiring a read lock and the operation is resumed by a cohort, the concurrency control algorithm no longer has knowledge of the read lock. As a result, inconsistencies may arise. Concurrency control algorithms of this sort are generally forced to abort operations and restart them if a failure occurs. This restriction can be removed from most algorithms simply by distributing concurrency control information (here, information about locks) to the cohorts. At first sight, distributing information about read locks may seem unreasonably expensive, but we have developed a scheme by which such information may be piggybacked on messages required for checkpointing and failure detection. This "lazy propagation" of information is based on the fact that information about a read lock is not significant until the activity holding the lock attempts to modify other data items, or some other activity attempts to update the locked data item. Details of this scheme are given in [Bir84b]. Designing a concurrency control algorithm that performs efficiently in a resilient object environment raises many difficult problems. In [Bir84a], we develop a novel concurrency control algorithm, which addresses these problems.

[3] At the moment a new checkpoint is added, both it and the old checkpoint are valid. Hence if the coordinator fails in the course of establishing a checkpoint, a cohort resumes from the new checkpoint if it has received the entire checkpoint information, and from the previous one, if not.

3. Improving performance

The number of checkpoints that have to be established depends on the extent to which results are retained. If objects retain results until the calling activity terminates, only one checkpoint, the initial checkpoint, is required in each activity. Even if results are not retained this long, checkpoints can be avoided in certain situations. If an action is idempotent, a cohort can re-execute it even if it was already executed, or if an object supports the ability to abort an action, a cohort can first issue an abort and then execute it[4]. Thus it is not necessary for the coordinator to establish a new checkpoint before executing an action that would cause the results of an idempotent or abortable action to be no longer retained.

There is a trade-off between the cost of establishing checkpoints and the cost of tolerating failures. If the number of checkpoints is reduced, fewer broadcasts are made. Communication costs are lowered and execution is faster. On the other hand, more operations have to be re-executed if a site fails. Eliminating checkpoints thus leads to a more efficient implementation in an environment where failures do not occur often. If the expected failure rate is high, it might be advantageous to establish checkpoints more often than actually required for resiliency. Much analytical work has been done on determining the optimal checkpoint frequency [Tou84], and it is possible for the coordinator to use such an algorithm to decide when to establish additional checkpoints.

Another alternative is to allow the user to provide "hints" as part of the object specification, indicating when it might be desirable to establish extra checkpoints. These additional checkpoints need not be established at all the cohorts: the coordinator could instead establish checkpoints frequently at one or two cohorts, and less often at the others, on the basis of a hierarchical ordering of the cohorts. If the coordinator fails, a cohort at which frequent checkpoints have been established becomes the new coordinator, and it does not have to back-out a great deal. If a number of sites fail simultaneously, it may be necessary for a cohort with an earlier checkpoint to take over, but this will happen less often than a single-site failure. Such a hierarchical method of establishing checkpoints reduces the message traffic considerably, while still preserving the advantages of frequent checkpointing.

Further optimizations can be made when a single version concurrency control algorithm is used. A single version algorithm has the property that if an object-data item is written upon, the old value is no longer available. The concurrency control algorithm used in *ISIS* satisfies this property. This is also true of

[4] We assume that aborting an action that has not been executed has no effect.

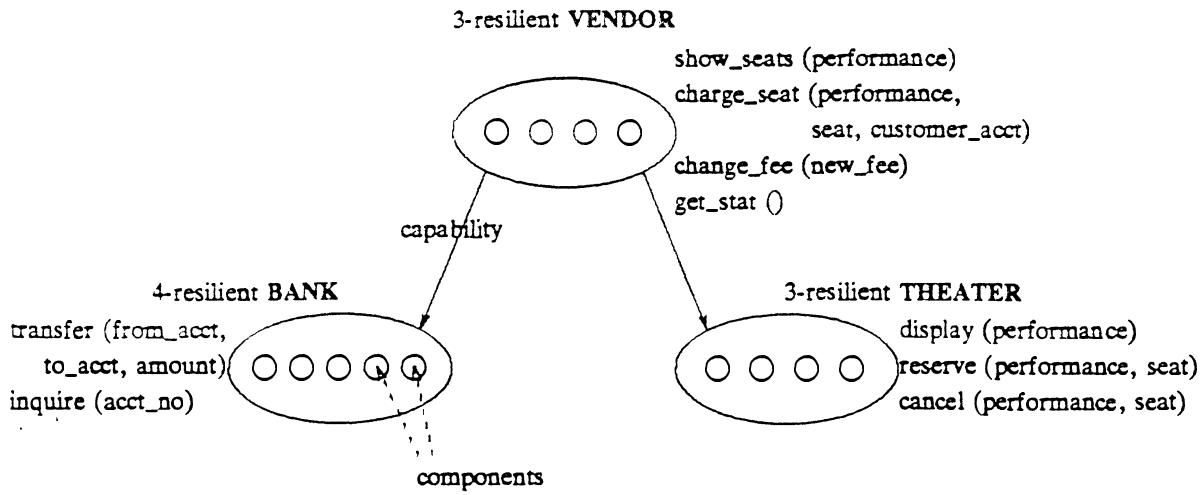

Figure 4.

a quorum based algorithm [Gif79, Tho79], for although different versions may exist at different sites, the quorums are so chosen that the result of a read is always the latest version written. In effect, only the latest version is accessible at any time. When a single-version concurrency control scheme is used, each site has the same view of the object-data. In particular, if the coordinator reads an object-data item and fails without altering it, the new coordinator reads the same value[5].

Assuming this behavior, the coordinator need not transmit the values of all the temporary variables and the object-data at each checkpoint. For example, no object-data values need be transmitted at the initial checkpoint, since the cohorts have the same view of the initial data. The coordinator can carry out all actions on temporary variables and on object-data locally, and when a new checkpoint is established, it need transmit only those values altered since the last checkpoint. Certain concurrency control methods, however, require that whenever an update is made on a replicated data item, it is made on more than one copy[6]. In this case, the value at a checkpoint made before an update will not be available to the cohorts after the update, unless it was stored as part of the checkpoint information. So, each time a cohort is required by the concurrency control algorithm to update its copy of an object data item, it checks whether an old copy exists in the checkpoint information, and includes one if not.

If results are retained until an activity terminates, a nested operation requires only one checkpoint to be established at each level. Assuming a single version concurrency control scheme, this can be reduced even further. If the activity-ID is passed down by the calling activity instead of being generated by the coordinator of the called activity, it is unnecessary to inform the cohorts of the activity-ID, except at the top-most level. Only one checkpoint is then needed for the whole nested operation. If the coordinator of an activity at the top-most level fails, a cohort takes over as usual. If a lower level coordinator fails, the remote procedure call mechanism reissues the call to another site, which now acts as coordinator and restarts the activity using the same activity-ID. The new coordinator issues operations under the same action-ID's as the failed one did, hence external actions already executed by the previous coordinator are not re-executed. Thus, the number of broadcasts made is just what is needed to maintain consistency, to retain results, and for one top-level checkpoint. If a concurrency control algorithm used does not release locks until the commit point (as is true of the default algorithm used in *ISIS*), it is not necessary to do a broadcast to retain the results of a read-only transaction: the values could be read again if a site fails. Finally, some of the broadcasts that have to be made can be merged together. For example, the distribution of retained results to the cohorts can be combined with the commit protocol required for con-

[5]This is not true when so-called *multiple-version* concurrency control mechanisms are used [Ber82]. In such settings, the new coordinator might read a different version, corresponding to a different serialization order.

[6]This is required when a quorum mechanism is used. With two-phase locking, write locks must be obtained at all the sites, but having locked a data item, the coordinator can update it locally. It is necessary to transmit its value to the cohorts only at checkpoints and at the end of the operation.

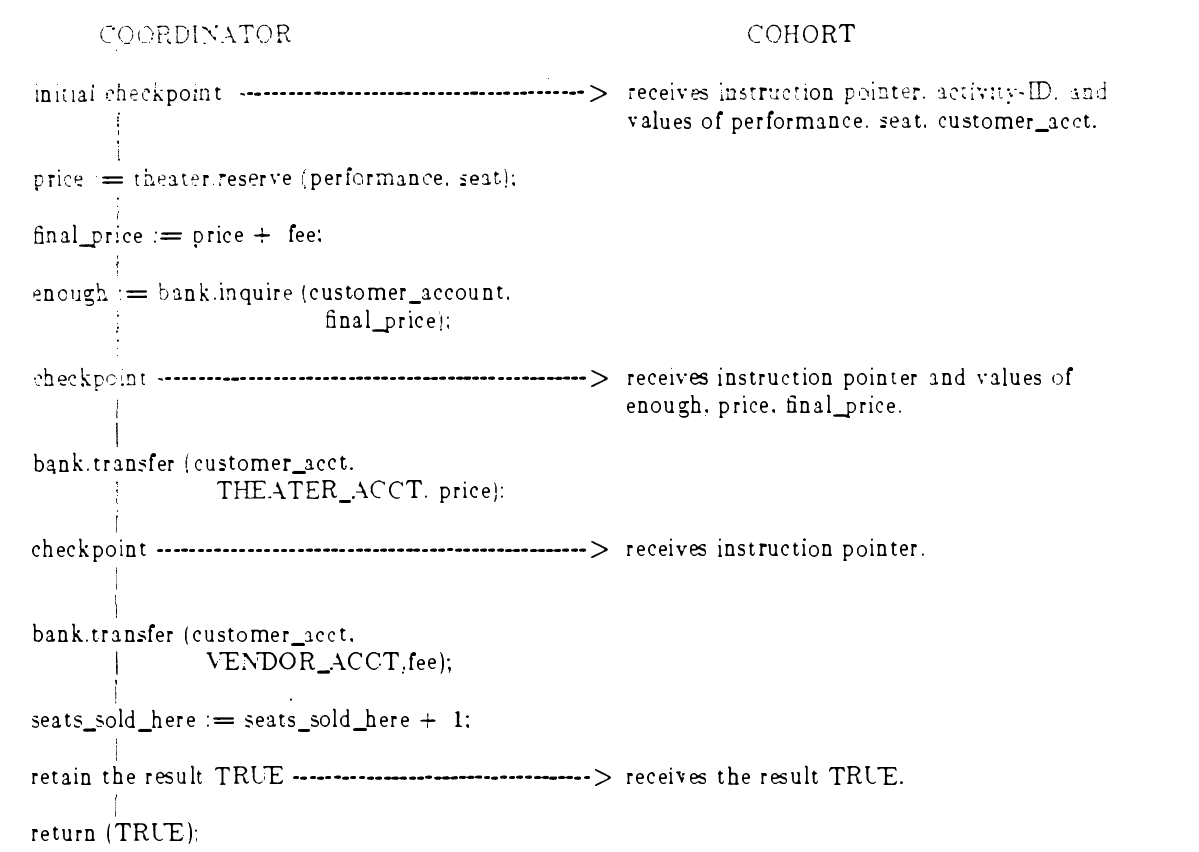

Figure 5(a).

currency control.

4. Example

We now present an example of the use of this technique. For the sake of clarity, we have limited the functionality of the objects described; actual objects would often support a much larger range of functions. Fig. 4 depicts three resilient objects: *vendor*, *bank*, and *theater*. *Vendor* reserves theater tickets and charges them to a customer's bank account. It supports the operations *show_seats*, which displays the seats available for a given performance, *charge_seat*, which charges the cost of a seat to a customer's bank account, and *get_stat* to get statistics. *Vendor* calls *bank* to transfer money between accounts, and calls *theater* to reserve the seats.

Bank has the operations *transfer*, which transfers a sum of money from one account to another, and *inquire*, which checks whether the balance in an account is large enough to cover a withdrawal.

Theater supports three operations : *display*, *reserve*, and *cancel*. *Display* displays the seats available for a particular performance. *Reserve* reserves a seat and returns its cost if the seat is available, and returns the value UNAVAILABLE if not. The operation *cancel* cancels a reservation. For increased concurrency, *display* does not lock the data it reads; it is therefore possible for a seat that was displayed as available to be unavailable when a reservation is attempted (a sad comment on real life!). *Charge_seat* takes this into account.

The specification of *vendor* was given in Fig. 1. The object-data are the variables *seats_sold_here* and *fee*. *Charge_seat* calls *reserve* in *theater* to reserve a seat, and transfers its cost from the customer's bank account to the theater's account. It also transfers a fee from the customer's account to the vendor's account. If the balance in the customer's account is insufficient to cover the transfers, the reservation is canceled. The other operations are self-explanatory.

Let us consider the actions taken within the object *vendor*, when it is invoked to perform *charge_seat*. We assume that the seat requested is available and that the customer's bank balance is sufficient to cover its cost. Fig. 5(a) shows the execution when the value of n, the number of results retained by *bank*, is 1, and Fig. 5(b) when n is 2. In the latter case, a checkpoint is avoided. Fig. 5(c) dep-

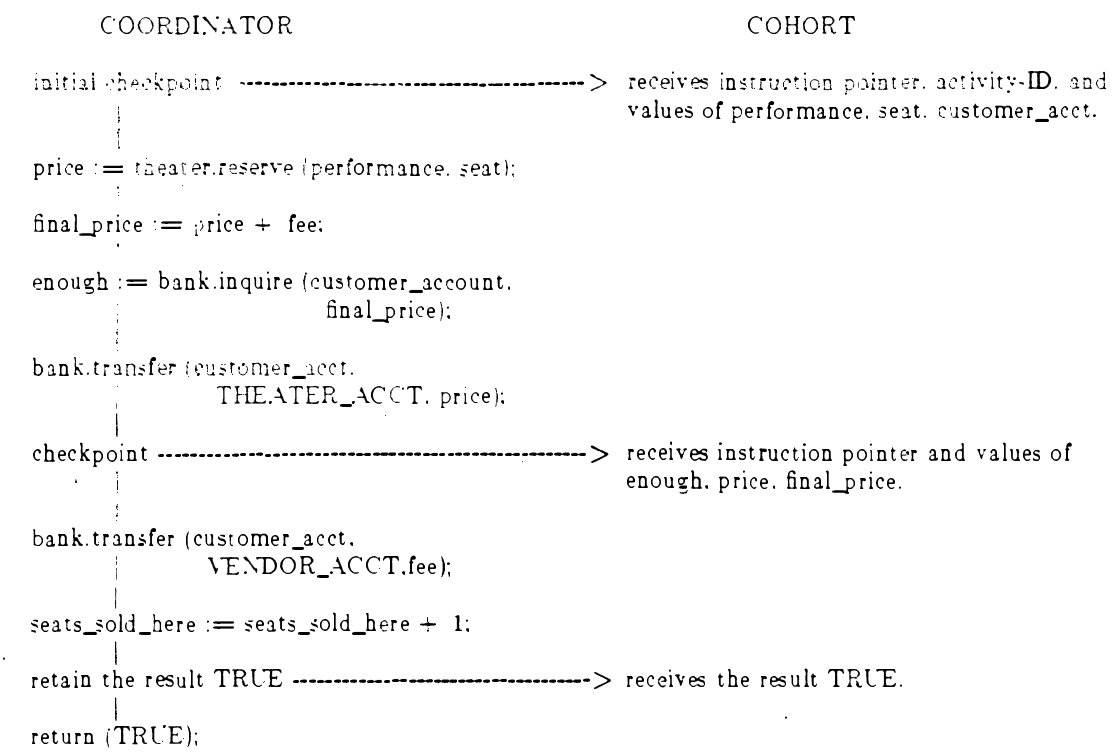

Figure 5(b).

icts an execution where the coordinator fails. Again, we assume that n is 2.

5. Extension to total failure

We have shown how a k-resilient object ensures that its operations are completed if up to k sites fail. If more than k sites fail, we may desire that the execution of an operation block until one or more of the sites recover, and then continue. To achieve this, the cohorts write all checkpoint information and retained results on non-volatile storage [Bay80]. When a site recovers from a total failure, it runs a protocol to determine whether it was the last site to fail [Ske83]. When the last site to fail recovers, it becomes the coordinator and executes forward from the last checkpoint. It uses the sites that have recovered before it as cohorts.

6. Applications

The technique described in this paper is now being implemented in *ISIS* for the support of resilient objects. *ISIS* permits a user to specify objects, which are then implemented resiliently. *ISIS* will also provide the user with a number of resilient base object types, which can be used to construct more complex objects.

In addition to achieving forward progress in operations, this mechanism permits *ISIS* to support a generalized abort facility. In other systems, an abort is implemented by saving old values of data items when modifications are made, and restoring them when an operation aborts. This approach is not very general, particularly if operations include external actions like "move a robot arm 3 inches". The effects of such operations can be nullified only by executing a sequence of "inverting" operations, called the *undo algorithm* in *ISIS*. *ISIS* permits the user to make any operation abortable by specifying an undo algorithm for the operation. Such an undo mechanism would be meaningless unless the execution of the undo algorithm is guaranteed to progress to completion despite failures, otherwise an operation may be incorrectly or partially undone. The scheme described in this paper is used within *ISIS* to provide these guarantees.

The fact that operations are guaranteed to proceed to completion allows greater flexibility in concurrency control. In some concurrency control algorithms, locks on data items can be released before an operation reaches its commit point. This increases the level of concurrency, but introduces the problem of "cascading aborts" : if an operation aborts for any reason, all operations that have read data written by it must be tracked down and aborted. As a result, this type of concurrency control is likely to be too

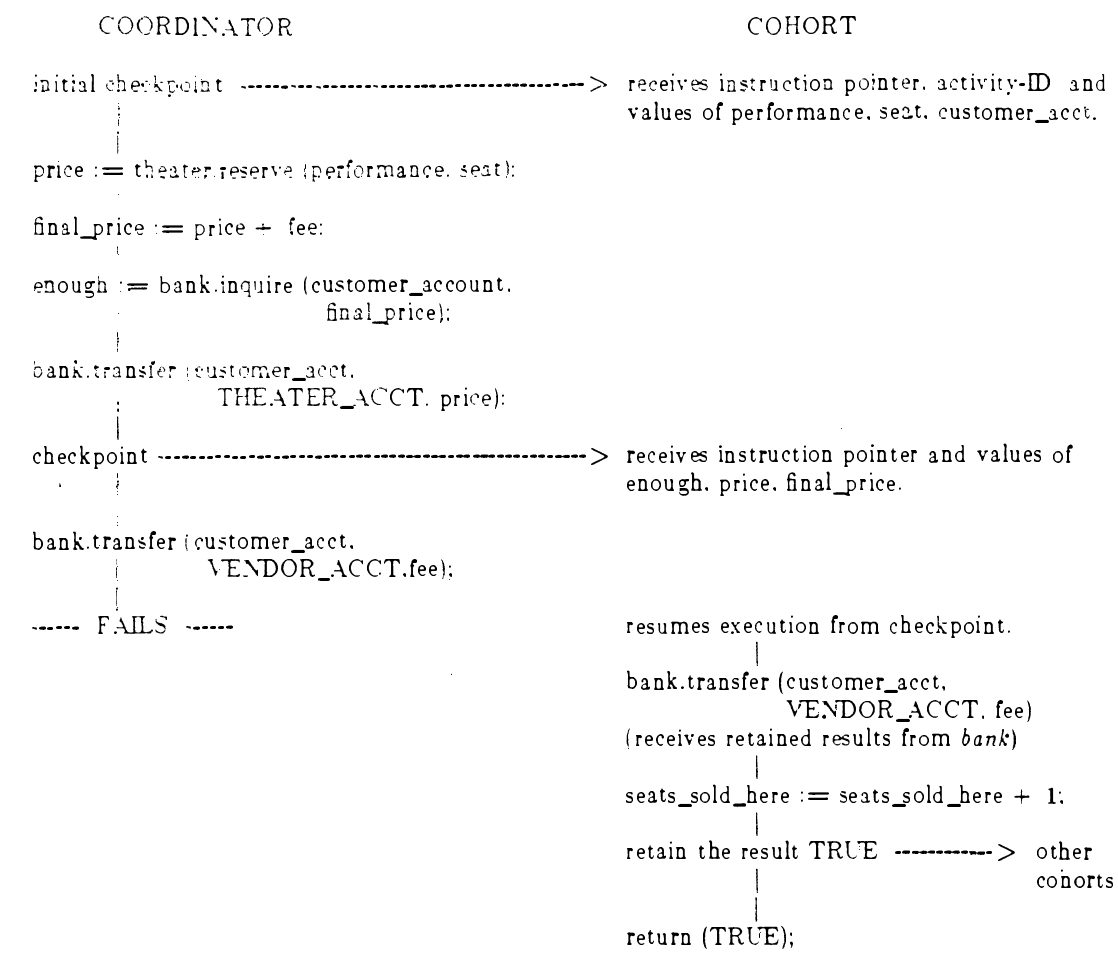

Figure 5(c).

expensive to implement. With the forward progress mechanism proposed here, operations do not abort because of site failures, and the use of such a concurrency control scheme could be justified for applications that are deadlock-free and in which explicit aborts do not occur. We are developing a concurrency control algorithm that achieves greater concurrency than with two-phase locking by exploiting this behavior and other properties of a distributed object-based system like *ISIS*. This will be discussed in a forthcoming paper [Bir84].

7. Conclusions

We have discussed a method of obtaining a resilient implementation of an object from a non-resilient specification using checkpoints and retained results. We have also shown how to reduce the communication overhead so that the implementation is efficient enough to be viable. The approach brings the implementation of a distributed fault-tolerant system within the realm of a relatively unsophisticated programmer, freeing him or her from the burden of implementing complicated synchronization mechanisms. It is being implemented in *ISIS*, a system that supports resilient objects. We expect that systems which provide very "high-level" services, like *ISIS*, will become indispensable as distributed processing becomes the most prevalent mode of computing.

8. Acknowledgements

We would like to acknowledge the many contributions which D. Skeen has made to the work presented in this paper. Fred Schneider read an earlier version of this paper, and this version has benefited immensely from his suggestions.

9. References

[All83] Allchin, J., McKenty, M. Synchronization and Recovery of Actions, to appear: *Proc. 2nd Principles of Distributed Computing*, Toronto, 1983.

[Bay80] Bayer, R. *et al.* Parallelism and recovery in a database system. *Trans. on Database Systems* 5, 2 (June 1980), 139-156.

[Ber80a] Berstein, P., Shipman, D., Rothnie, J. Concurrency Control in a System for Distributed Databases (SDD-1). *Trans. on Database Systems* 5, 1 (March 1980), 18-51.

[Ber80b] Bernstein,P., Shipman, D. The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1). *Trans. on Database Systems* 5, 1 (March 1980), 52-68.

[Ber81] Bernstein, P., Goodman, N. Concurrency Control in Distributed Database Systems., *ACM Computing Surveys* **13**, 2 (June 1981), 185-222.

[Ber82] Bernstein, P., Goodman, N. Concurrency Control Algorithms for Multiversion Database Systems. *Proc. First Principles of Distributed Computing*, Ottawa, 1982, pp. 209-215.

[Ber83] Bernstein, P. A. and Goodman N. The Failure and Recovery Problem for Replicated Databases. *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, Montreal, 1983, pp. 114-122.

[Blr84a] Birman, K.P., Joseph, T.A., Raeuchle, T. Extending Resilient Object Types Efficiently, *Cornell University, Technical Report. Second GID/NTG/GMR Conference on Fault-Tolerant Systems*, Bonn, Sept. 1984.

[Blr84b] Birman, K. Replication and Availability in ISIS. Department of Computer Science, Cornell University. *Forthcoming.*

[Bor83] Borg, A., Baumbach, J., Glazer S. A Message System Supporting Fault Tolerance, *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, (Oct. 1983), 90-99.

[Ich79] Ichbiah, J. D. *et al.* Preliminary ADA reference manual, *SIGPLAN Notices* **14**, 6 (June 1979).

[Gif79] Gifford, D. Weighted Weighted Voting for Replicated Data, *Operating Systems Review* **13**, 5 (Dec. 1979), 150-159.

[Kun81] Kung, H., Robinson, J. An Optimistic Method for Concurrency Control, *Transactions on Database Systems* **6**, 2 (June 1981), 213-226.

[Lis74] Liskov, B. H. and Ziller, S. N. Programming with Abstract Data Types, *Proc. ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices* **9**, 4 (April 1974), 50-59.

[Lis81] Liskov, B. On Linguistic Support for Distributed Programs, *Proc. Principles of Programming Languages*, 1981, pp. 53-60.

[Lis82] Liskov, B, Scheifler, R., Guardians and Actions: Linguistic Support for Robust, Distributed Programs, *Proc. Principles of Programming Languages*, 1982, pp. 7-19.

[Lyn82] Lynch, N. Concurrency Control for Resilient Nested Transactions M.I.T. EECS Department, August 1982.

[Mos81a] Moss, J.E., Nested Transactions and Reliable Distributed Computing, *Proceedings Second Symp. on Reliability in Distributed Software and Database Systems*, 1981, pp. 33-39.

[Mos81b] Moss, J.E., Nested Transactions: An Approach to Reliable Distributed Computing, Ph.D. thesis, M.I.T. Dept. of EECS, available as M.I.T. Lab. for CS Tech. Report 260, April 1981.

[Nel81] Nelson, B. Remote Procedure Call, Ph.D. thesis, Carnegie Mellon University, Computer Science Tech. Report CS-81-119, Pittsburgh, PA, 1981.

[Sch83] Schlichting, R.D., Schneider, F.B. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *TOCS*, August 1983.

[Ske83] Skeen, D. Determining the Last Process to Fail., *Proc. Symposium on the Principles of Database Systems*, Atlanta, Georgia, March 1983.

[Tho79] Thomas, R. A Majority Consensus Approach to Concurrency Control, *Transactions on Database Systems*, **4**, 2 (June 1979).

[Tou84] Toueg, S. and Babaoglu, O. On the Optimum Checkpoint Selection, to appear in *SIAM Journal on Computing*, Vol. 13, No. 3, Aug 1984.

[Wir77] Wirth, N. Modula: A Language for Modular Multiprogramming, *Software Practice & Exp.*, **7**, 1 (1977).