An Analysis of Borrowing Policies for Escrow Transactions in a Replicated Data Environment

Akhil Kumar
Graduate School of Management
Cornell University, Ithaca, NY 14853.
(Bitnet: akhil@cmlgsm)

Abstract
In a replicated data environment, escrow-based techniques can reduce the transaction processing overhead dramatically for transactions that access and update aggregate fields such as the quantity of an item in stock or the number of seats available on an airplane. In this paper, three issues are addressed that were omitted in previous work in the context of escrow-based methods. First, we show that full serializability can be guaranteed for escrow transactions by means of repeat semantics, i.e., a transaction which fails is rerun in a special mode. The second objective is to describe a crash recovery protocol consistent with our concurrency control scheme. Finally, we analyze, by means of a simulation model, several different borrowing policies for escrow transactions and identify the best one.

1 Introduction
Conventional methods for replicated data transaction management are expensive because more than one site is required to form a quorum in order that an update transaction may run. Some examples of such methods are: majority consensus [THOM79], quorum consensus method [GIFF79], and dynamic voting [JAOJ87]. These methods do not make any assumption about the special semantics of the transactions or the nature of the underlying data objects which are accessed and updated. Several researchers (e.g., [GRAY81, REUT82, ONEI86]) have noted that if the data objects on which transactions operate are aggregate fields, and only restricted operations are allowed on them, then specialized techniques for transaction management may be devised. These techniques exploit the commutativity property whereby two transactions that perform commutative operations (say, add and subtract) can run in any order and yet yield the same final result. Other kinds of specialized techniques are discussed in [GARC83, BLAU85, SARI85].

In a previous paper [KUMA88], we extended such techniques to a replicated data environment and showed how both special and general transactions can be treated in a common framework. Here we shall restrict ourselves to issues that pertain to special transactions alone. The special class of transactions we shall discuss are not purely commutative transactions; rather, they commute subject to lower and/or upper bound restrictions (or integrity constraints) on the objects on which they operate. For example, two withdrawal transactions on the same bank account commute if the final balance after both of them complete is positive; else, they do not commute. Such transactions occur frequently in real application environments like banking and airlines. On the other hand, purely commutative transactions seldom occur in real applications and, hence, are less interesting.

The technique proposed in [KUMA88] for handling special transactions consists of allocating the amount available in an aggregate field across several escrow pools, one at each site where a copy of the file resides. A withdrawal transaction would first verify that the required amount was available in the local escrow pool, and if not, would attempt to borrow from the escrow pools at other sites. If the amount to be withdrawn is available in the local escrow, the transaction can commit locally, and then spool a message to all other sites. The message consists of an object identifier, transaction type, and the amount of the transaction, and each receiving site would run it as a local transaction. All locks on the escrow fields are short-term locks, held only for the duration of an operation, while data object locks are held for the length of a transaction. The advantage of this scheme is that a large number of transactions would not need to exchange any messages with other sites, and, therefore, would run very fast. Moreover, read operations get a read lock on the local copy of an object and return its value.

However, a major shortcoming of this technique is that strict serializability is not preserved. This has two implications: (1) Two transactions running...
concurrently may both fail, while if they ran serially one may succeed, and (2) read operations return out of date answers. Consider an object representing the balance in a bank account, replicated at two sites. Further assume that the account balance is $1000 and is equally divided between the escrow pools at the two sites, each containing $500. Moreover, consider two concurrent transactions, T1 and T2 running as follows:

T1(at site 1): withdraw $1250
T2(at site 2): withdraw $750

T1 starts first, and appropriates $500 from its local escrow pool (leaving 0 there) and another $500 from the escrow at site 2 (leaving 0 there, too). T2, running concurrently, is unable to obtain $750 from the two escrows. Therefore both transactions fail. On the other hand, if T1 and T2 ran serially (in any order), T1 would fail and T2 would succeed, and this means that the algorithm needs to be modified to ensure that serializability is preserved.

The second issue relates to crash recovery. The write ahead logging (WAL) algorithm is a standard and efficient technique for crash recovery [GRAY78, REUT84, BERN87]. The WAL algorithm, however, assumes that transactions are executed according to the two phase locking protocol which requires that all objects be locked before access and that these locks be held until the end of the transaction. It is, therefore, important to note that in the absence of two-phase locking, the correctness of WAL cannot be established. Since our algorithm does not observe the two-phase locking protocol strictly (locks on escrow field are only short-term), the standard WAL algorithm will require modification.

The organization of this paper is as follows. In Section 2, we describe modifications to our basic escrow technique in order to guarantee serializability. Next, Section 3 discusses changes to the WAL algorithm in order to make it consistent with the concurrency control scheme. The remainder of the paper is devoted to an analysis of borrowing policies using a simulation model. The policies and the model are described in Section 4, while the results are presented in Section 5.

2 Concurrency Algorithms

In section 2.1, we shall review our basic algorithm for running escrow transactions, and then, in Section 2.2, show how it can be modified to ensure serializability.

2.1 Basic Escrow Method

Here we describe the basic escrow method for performing increment and decrement operations on replicated fields that contain aggregate data. It is assumed that constraints exist on the minimum value that each object may assume, while no constraints exist on its maximum value. Each object is assumed to be replicated at n sites, and an escrow field is associated with each copy. The escrow pools at different sites are initialized such that each contains the same quantity and they add up to the value in the corresponding aggregate field. The main steps of the algorithms for performing increment and decrement operations are listed below, and a brief discussion follows.

Algorithm Escrow-Increment(amount: c){
write-lock local copy of object.
increment local copy by c.
(other steps)
(at commit time) increment local escrow field by c.
unlock local copy of object.
spool message to all other sites: (object id, transaction type, amount).
}

Algorithm Escrow-Decrement(amount: c){
write-lock local copy of object
If local escrow > c then
reduce local escrow by c
else
try to borrow c from local and remote escrows.
If borrowing succeeds then
decrement local copy of object by c.
If borrowing fails then
return any borrowed amounts to escrows.
write message ("transaction failed"), and abort.
}

The following points should be noted regarding the above algorithms. First, the increment operation is always successful since lower bound constraints exist on the value of the object. Secondly, all updates to escrow fields require only short-term locks, i.e., locks are held only for the duration of the update step. Thirdly, a site which receives a message from another site executes the following algorithm in order to install on its own copy the updates made to an object elsewhere.

Algorithm Receiving Site(object id:o, transaction type:t, amount:c){
write-lock object o.
If t is increment then
Increment \( o \) by \( c \)
else
Decrement \( o \) by \( c \)
Unlock \( o \).

Note that for simplicity, in all three algorithms above, only single object updates are shown. The extension to the multiple object update situation is straightforward. The spool message in that case will consist of a sequence of (object id, transaction type, amount) triples. Moreover, upper bound constraints may be implemented in a similar way; a separate escrow field would have to be maintained for this purpose. Finally, at periodic intervals, the quantities in the various escrow pools are reallocated equally by running a special transaction.

2.2 Modifications

In this section we discuss the modifications to the basic escrow method in order to guarantee serializability. The modifications are based on repeat semantics, and the central idea is that any transaction that fails must run again in a special mode. In this mode, the transaction must first obtain read locks on all copies of the object and, thereby, prevent another transaction from performing any simultaneous updates. Therefore, the algorithm is written as:

Algorithm Sound{
execute algorithm escrow-decrement(\( c \)).
If transaction fails, then
run Algorithm Special Mode(\( c \))
}

Algorithm Special Mode(\( c \)){
read-lock all copies of object.
exact-value of object = sum of all escrow fields.
If exact-value < \( c \) then
confirm transaction fails, and abort.
else{
borrow amount \( c \) from local and remote escrow
write-lock local object copy
decrement local copy by \( c \)
release locks on remote object copies.
}
... (other steps)
(at commit) unlock local copy of object.
confirm end of transaction.
spool message to all other sites: (object id, transaction type, amount).
}

This modification now guarantees serializability under all situations. The correctness of this algorithm may be argued simply as follows. A transaction that fails after borrowing some amount from its local and remote escrow pools will return the borrowed quantity back. On the other hand, if a transaction succeeds it will never return anything to escrow. By temporarily appropriating from escrow, a failed transaction may cause another concurrent transaction also to find that insufficient amount is available in escrow and to fail. It is possible that if the two transactions ran serially one of them would have succeeded. Clearly, this can result in lack of serializability. Forcing all failed transactions to rerun in special mode (by locking all copies) ensures that serializability is maintained.

It is evident that a greater overhead must be incurred in rerunning failed transactions. If few failures are expected then this should not pose a problem. On the other hand, in some situations the overhead could be high; for instance, in an airlines reservation application, after a flight is sold out. This situation may be corrected by making another modification to the above algorithm. A flag is associated with each object, and a transaction which fails and finds that the exact value of the object is equal to its minimum permissible value, would set this flag at all sites. Moreover, any transaction that performs an increment operation and finds the flag set, would reset it again at all sites. Finally, a transaction that runs in special mode would first check if the flag is set, and if so, it may immediately reconfirm the failure. The revised algorithm, Special Mode New, is as follows.

Algorithm Special Mode New(\( c \)){
If local flag is set then
confirm failure and abort.
read-lock all copies of object.
extact-value of object = sum of all escrow fields.
If exact-value < \( c \) then {
If exact-value = 0 then set all flags.
confirm transaction fails, and abort.
}
else{
borrow amount \( c \) from local and remote escrow
write-lock local object copy
decrement local copy by \( c \)
release locks on remote object copies.
}
... (other steps)
... (at commit) unlock local copy of object.
confirm end of transaction.
spool message to all other sites: (object id, transaction type, amount).
}

We now turn to the issue of read serializability. Basically, we propose to have two types of read operations: a cheap read and a special, more expensive read. The cheap operation would be local, and would
obtain a read lock on the local copy of the object and return its value. The returned value can possibly be out of date if a concurrent update has been executed at another site. Therefore, to obtain an exact value, a read transaction must run in a special mode. A transaction running in this mode first obtains read locks on all the copies of the object. The exact value of the data field is now computed as the sum of the individual escrow fields at all sites, as in Algorithm Special Mode above. Since a transaction must have a read or write lock on at least one copy of an object before accessing any escrow field associated with an object, the special read operation precludes the possibility of such concurrent access by read-locking all copies.

The advantage of having two types of read operations is that the more expensive one can be reserved for special situations, while the cheaper read may be used where slightly out of date data is tolerated. The underlying assumption is that the special read would be required only rarely. Moreover, if the spooler program (for spooling messages to other sites) at each site is assigned a sufficiently high priority then the length of out-of-datedness would typically be small.

3 Recovery Algorithm

The principle behind the write ahead logging (WAL) technique [GRAY78] is that before- and after-images of all modified objects or records are written into a log and this log must be moved to disk or other stable storage before the corresponding page on which the object or record resides. The protocol implicitly assumes that concurrent transactions observe the two phase locking protocol in order to guarantee serializability. In the event of a crash, the standard WAL recovery algorithm is run independently at each failed site as follows:

1. (in the backward scan) Scan beyond the most recent checkpoint, and undo all transactions (both committed and uncommitted) by installing successive before-images, until all active transactions at the time of crash are undone.

2. (in the forward scan) Redo each committed transaction by executing (operation, amount) on escrow fields for each entry in the log that corresponds to a committed transaction. For non-escrow fields successive after-images are installed.

This algorithm is more expensive than the original WAL because it requires all transactions to be undone in the backward pass from the end of the log to the most recent checkpoint. The other difference is that the committed transactions must be redone based on the operations that have been logged, not by installing after-images.

The next section turns to an evaluation of various borrowing policies.

4 Experimental Set-up

The algorithms of Section 2 are based on sites borrowing from their local escrow pools or from the escrow pools at remote sites in order to perform decrement transactions. No specific policies for borrowing were included there and we address that subject in the remainder of the paper. Four borrowing policies were studied using a simulation model. In Section 4.1 we describe the policies, and then explain the simulation model in Section 4.2.

4.1 Borrowing Policies

The escrow method is based on each site maintaining an escrow pool and borrowing from other sites if the quantity available in its local pool falls short of the required amount. Several borrowing policies are possible and the issue of choosing an optimal policy has not been addressed previously. We define an optimal borrowing policy as one which minimizes the total message cost. We considered two factors in designing a borrowing policy: the method for choosing one or more lender sites and the amount to be borrowed. By combining these two factors, four policies were constructed as shown in Table 1.
A policy is represented by two letters, and each letter may take two values; thus, there are 4 policies. The first letter denotes the scheme for determining the amount to borrow, represented by an N or E. A borrowing site may either borrow the exact amount it needs to complete the current transaction or borrow an amount such that the final balance at the lender and borrower sites are equal after completing the current transaction. For example, consider two sites, 1 and 2, where the escrow pools contain 0 and 1000, respectively. If a decrement transaction at site 1 creates a need for 400 units, according to the first scheme site 1 would borrow exactly 400 units, while by the second scheme it would borrow 700 so that after completing the transaction, both sites 1 and 2 have 300 units left. The second letter in the policy name refers to the method for selecting lender sites. The two alternative methods are represented by an S or R. The first denotes selection of lender sites in a pre-specified order, while the second one refers to a random site selection order.

### 4.2 Simulation Model

A simulation model was used to evaluate these four policies. We assumed a banking environment where deposit and withdrawal operations are performed on customer accounts, however, the results would apply to any other environment where increment and decrement operations are performed on aggregate fields. Moreover, although here we assume bounds exist on the minimum value of a field, similar policies can be used, with minor modifications, if upper bounds are to be enforced. Finally, even though a minimum balance requirement of 0 is assumed, this does not cause any loss of generality since any arbitrary lower bound constraint may be imposed. We shall first describe our simulation model, then turn to the results of the experiments and finally draw conclusions from them.

Table 2 lists the important parameters of the simulation. The deposit and withdrawal amounts are drawn from uniform distributions. The distribution for deposits varies between 250 and 500, while that for withdrawals varies between 50 and 100. Withdrawals occur 5 times as frequently as deposits and, therefore, the total amount of deposits is equal to the sum of all withdrawals, on average. It turns out that the total balance, however, rises gradually over time because withdrawals that would result in a negative final balance are rejected while deposits are always allowed.

The number of sites where transactions originate is assumed as 7. Initially, it is assumed that each object is replicated at all 7 sites. Subsequently, the number of copies is reduced from 7 to 1, eliminating 1 copy at a time. The reliability of each site, i.e., the probability that a site is independently up, is initially set at 1 and later varied in the range 0.8 to 1 in order to examine whether it impacts the best borrowing policy. The default value of the starting or initial balance is 1500, and this again is varied over a range of values to see how it affects the best policy.

There are two performance criteria of interest. The first criterion is *avg-msgs*, the average number of round-trip messages per transaction, while the second one is *local-percent*, the percentage of transactions that run locally. The cost of sending a message from any site i to another site j is assumed constant. A transaction which can run locally costs 0 messages, while a transaction which communicates with k sites costs k round-trip messages.

The simulator runs as follows. Initially the escrow value of each aggregate field at every site where a copy exists is set to (field value/c), where c is the number of copies. Next, transactions are generated with equal frequency at all sites. The information produced by the simulator is: site number, transaction type, and amount. As stated in Section 2, a deposit transaction may always run locally and does not incur a communications cost. It write-locks the field being updated and increments it by the deposit amount. At transaction commit time, the escrow field is also incremented by the deposit amount, and a message is sent to all other sites, indicating the

#### Table 1: Alternative Borrowing Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td>borrow exact qty needed;</td>
</tr>
<tr>
<td></td>
<td>lender selection pre-specified</td>
</tr>
<tr>
<td>NR</td>
<td>borrow exact qty needed;</td>
</tr>
<tr>
<td></td>
<td>lender selection random</td>
</tr>
<tr>
<td>ES</td>
<td>final escrow balances equal;</td>
</tr>
<tr>
<td></td>
<td>lender selection pre-specified</td>
</tr>
<tr>
<td>ER</td>
<td>final escrow balances equal;</td>
</tr>
<tr>
<td></td>
<td>lender selection random</td>
</tr>
</tbody>
</table>

#### Table 2: Main Parameters of the simulation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min-Bal</td>
<td>Minimum balance</td>
<td>0</td>
</tr>
<tr>
<td>Dep-Distrn</td>
<td>Distribution of deposits</td>
<td>U(250,500)</td>
</tr>
<tr>
<td>Wdl-Distrn</td>
<td>Distribution of withdrawals</td>
<td>U(50,100)</td>
</tr>
<tr>
<td>W/D ratio</td>
<td>Withdrawal to deposit ratio</td>
<td>5</td>
</tr>
<tr>
<td>num-sites</td>
<td>Number of sites</td>
<td>7</td>
</tr>
<tr>
<td>num-copies</td>
<td>Number of sites with copies</td>
<td>7</td>
</tr>
<tr>
<td>Reliability</td>
<td>Probability a site is up</td>
<td>1.0</td>
</tr>
<tr>
<td>init-bal</td>
<td>Initial balance</td>
<td>1500</td>
</tr>
<tr>
<td>avg-msgs</td>
<td>avg number of messages</td>
<td></td>
</tr>
<tr>
<td>local-percent</td>
<td>% local transactions</td>
<td></td>
</tr>
</tbody>
</table>

Note that the choice of an optimal policy for initially allocating amounts to escrow fields at various sites is another optimization issue, but we do not consider it here because we believe the borrowing policy is more important.
transaction type and the amount of deposit. A withdrawal transaction may run locally if the amount of withdrawal is less than the quantity in escrow. Otherwise, it must borrow from other sites according to the policy being evaluated.

Now we turn to describe the results of the experiments which were conducted.

5 Results of Experiments

With the above experimental setup, several experiments were conducted to evaluate the four policies listed above. In these experiments many parameters were varied in order to simulate the performance in different environments. These are discussed in the subsequent sub-sections.

5.1 Varying number of Sites

In the first set of experiments, the number of copies, num-copies was varied between 1 and 7, while all other parameters were set at their default values of Table 2. Num-sites, the number of sites where transactions originate was kept at 7. Sites where a copy of an object did not exist had to communicate with at least one other site each time they ran a transaction. At each value of num-copies and for each borrowing policy, the average number of messages was computed. Figure 1 is a plot of average number of messages versus number of copies, for all four policies, while Figure 2 is a similar plot displaying the percentage of local transactions on the Y-axis.

Figure 1 shows that all four policies perform better as num-copies increases, and ER is the dominant policy for all values of num-copies, except when it is 2. For one copy of the object, all four policies are identical. When num-copies is 2, Policies NR and NS, each cost 0.77 messages, and are very marginally better than ER and ES, each of which cost 0.78 messages. As num-copies increases above 2, ER and ES do consistently better than the other two policies for each higher value of num-copies. The gap between ER and ES is generally narrow except when num-copies is 7, in which case ER does appreciably better. It should also be noted from Figure 1 that for num-copies equal to 7, the gap between the best policy (ER) and the worst policy (NS) is 40%. Figure 2 shows that, under Policy ER, 90% of all transactions run locally in the 7-copy case, and it is the dominant policy in terms of the second criterion. For fewer than four sites, all the policies seem to perform very similarly on this criterion; however, beyond four sites, ER and ES perform much better than NR and NS policies. Clearly, the right choice of a borrowing policy has a major impact on both performance criteria.

These experiments illustrate that among the two elements of a borrowing policy, amount borrowed from different sites and the order in which potential lender sites are chosen, the former is more important. Consequently, the ER and ES policies performed considerably better than NR or NS. Moreover, since ER did better than ES, and NR than NS, it is evident that a random method for selecting a lender site is superior to selecting them in a pre-specified order.

5.2 Varying Site Reliability

In the next set of experiments, we varied the individual site reliabilities to see how it affects the various policies. The reliability, or probability that a site is up, was varied between 0.8 and 1.0, and the average message cost was recomputed by simulation. Num-sites was kept at 7 in all cases. The results are plotted in Figure 3 (an avg-msgs versus reliability graph) and Figure 4 (local-percent versus reliability graph).
Figure 3: Average messages versus reliability of sites

Figure 4: Percentage of local transactions versus reliability of sites

Figure 5: Average messages versus initial balance

5.3 Varying Initial Balance

In the third set of experiments, the initial balance was varied. A larger initial balance would be expected to affect message costs in two ways. First, assuming identical withdrawal and deposit distributions, a larger starting balance would increase the probability that a withdrawal transaction would run locally. Secondly, it would lower the probability of an overdrawn account. One would intuitively expect that both these factors would lead to a reduced average message cost as the initial balance increases.

In order to study the validity of this hypothesis and also the relative performance of the various policies, init-bal was varied over a range of values from 500 to 5000, while all other parameters were set to their default values of Table 2. Again, the average message cost was computed. The results are shown in Figure 5, an avg-msgs versus init-bal plot, and Figure 6, a similar plot with local-percent on the Y-axis. As expected, the average message cost falls as init-bal increases. The magnitude of the drop is approximately the same in each case. However, in percentage terms the drop is largest for policy ER (41%) and least for policy NS (27%). The percentage of local transactions is also consistently 5 to 10% higher for ER as compared to NS policy.

It is, therefore, evident that varying init-bal does not change the ranking of the various policies. Furthermore, the percentage gap between the best policy, ER, and the worst policy, NS, in terms of message cost, grows as init-bal increases. When init-bal is 500, this gap is 36%, while for an init-bal value of 5000 it swells to 48%.

5.4 Varying W/D Ratio

Finally, we varied W/D-ratio, the ratio of withdrawals to deposits, to study its effect on the behavior of the four policies. Basically, the W/D-ratio whose default value is 5, was varied from 3 to 8. In order to maintain the total amount of withdrawals equal to the total deposits, on an average, the distribution from which the deposit amounts were taken was varied appropriately, while the distribution of with-
drawals was kept fixed. Thus, with a $W/D$-ratio of 3, the deposit distribution was made $U(150,300)$, while for a $W/D$-ratio of 7, the corresponding distribution was $U(350,700)$. All other parameters were kept at their default values, and the average message cost was recomputed with each policy. The results appear in Figures 7 and 8, where avg-cost and local-percent respectively, are plotted against $W/D$-ratio.

These results demonstrate that the rank ordering of the four policies remains unaffected in terms of both performance criteria. Moreover, the best policy (ER) is sometimes almost 50% better than the worst policy (NS) in terms of average message cost. The absence of a monotonic relationship between $W/D$-ratio and the two performance criteria is not surprising since the average amount of each deposit and each withdrawal is also modified in the same proportion as the $W/D$-ratio. Besides, our objective here is to study the relative performance of the various policies and not the behavior of any individual policy.

### 5.5 Analysis of Results

It should be clearly noted that all four policies perform immensely better than a general scheme like majority consensus algorithm [TlOM79]. The majority consensus algorithm would result in a minimum message cost of 3.0 (with 7 copies) and 0% of the transactions would be local. The results of the experiments in this section illustrate that it is possible to do considerably better on both criteria by means of special techniques.

Several important parameters of the simulation model were varied to study the behavior of the four policies under widely different conditions. The results of the various experiments described above establish that the four policies evaluated may be rank ordered as follows:

$$\text{ER} \geq \text{ES} \geq \text{NR} \geq \text{NS}$$

To gain an intuitive understanding for this result, the following two points are useful. First, if a site in need borrows the exact amount it requires for its current transaction, then it is guaranteed that if the next transaction at that site is a withdrawal it would again have to borrow from another site. On the other hand, if in the first instance it had borrowed more than what was needed immediately, the extra addition to its escrow field would reduce the future need for communicating with other sites. Hence, a borrowing policy in which the final balances at the lender and borrower sites are made equal does better than the need-based policy.

Secondly, in a pre-specified scheme for site selection, a given site always borrows from the other sites in the same order. Clearly, a site which lends to another site might exhaust its resource and a borrowing site would be better off trying to borrow from a different site, the next time around. With a random
lender selection scheme, the probability that a given borrower site will choose the same site as the lender on two successive occasions is reduced. This explains why the ER policy is the best one. Moreover, it is also evident that, of the two factors that constitute a policy, the scheme for determining the amount to be borrowed is more critical than the order in which possible lender sites are chosen.

6 Conclusions

In this paper, we first discussed special techniques for guaranteeing serializability for escrow transactions on replicated data, and also described how a standard crash recovery protocol must be modified to make it consistent with the concurrency control scheme. These techniques are naturally useful because, as our experiments show, they enable a large fraction of transactions, in some cases 90%, to run locally.

The other important contribution of this paper is to demonstrate that the choice of a correct borrowing policy can have considerable impact on the communications overhead incurred by the algorithm, both in terms of the average number of messages and the percentage of transactions that run locally. Four policies, which are intuitively simple and also easy to implement, were evaluated using a simulation model. Large differences in their relative performance in terms of two criteria were observed and the ER policy was found to be consistently superior to the others. We found that the average number of messages for ER policy was, in several instances, 40 to 50% lower than for the worst policy, NS.

An interesting topic for future research would be to examine other policies and see if they would do better than ER. We are currently investigating some variations of the ER policy. The preliminary results show that the improvement in performance is small, while the complexity of implementation becomes much greater. Yet another related area of interest would be to study the effect of the initial allocation policy on message cost.

References


454