An Optimistic Algorithm for Consistent Replicated Shared Data

Orran Krieger and Michael Stumm
Dept. of Electrical Engineering
University of Toronto

Abstract

In this paper, we propose a new algorithm, TORiS, for implementing the shared data model. TORiS replicates data at each site and uses an optimistic approach for providing a coherent distributed shared data space. It is the use of optimistic techniques that sets TORiS apart from other previously proposed algorithms that implement the same model.

1 Introduction

Traditionally, communication among processes in a distributed system is based on the data passing model. Message passing systems and systems that support remote procedure calls adhere to this model. The data passing model is a logical and convenient extension to the underlying communication mechanism of the system. In this model, port or mailbox abstractions along with primitives such as send or receive are often used to achieve the inter-process communication. This functionality can also be hidden in language-level constructs, as is done with RPC mechanisms. In either case, distributed processes pass shared information by value.

In contrast to the data passing model, the shared data model provides processes in a distributed system with a shared address space. Application programs can use this space in the same way they use normal local memory; that is, data in the shared space is accessed through read and write operations. However, in the case of distributed shared data, processes on multiple processors in the system may access the shared space at the same time. As a result, distributed applications pass shared information by reference.

The primary advantage of the shared data model over the data passing model is the simpler abstraction provided to the application programmer, an abstraction the programmer already understands well. The access protocol used is consistent with the way sequential applications access data, allowing for a more natural transition from sequential to distributed applications. Consider, on the other hand, the data passing model. Since processes must explicitly use communication primitives and channels or ports, the data passing model forces programmers to be conscious of data movement between processes at all times. Also, since the data passing model deals with multiple address spaces, it is difficult to pass complex data structures; data that is passed between processes must be packed and unpacked. With a shared data facility, complex structures can be passed by reference. Another advantage of the shared data model is that it hides the remote communication mechanism from the processes, again, substantially simplifying the programming of distributed application. We have found the size\(^1\) of programs using the shared data model typically to be half the size of equivalent programs that use the data passing model.

1.1 Background

The advantages of the shared data model have made it the focus of recent study and prompted the development of various algorithms for implementing the shared data model \([2, 3, 4, 10]\). Several implementations have demonstrated that, in terms of performance, the shared data model can be competitive to the data passing model, even in loosely coupled distributed systems \([2, 10]\).

For example, Li's algorithm \([10]\) partitions the shared data space into pages and copies of these pages are distributed among the processors, obeying a single-writer-multiple-reader (SWMR) protocol. Pages that are marked "read-only" can be replicated and may reside in the memory of several processors, but a page being written to can reside only in one memory. This protocol is similar to the write invalidate algorithms used for cache consistency in shared-memory multiprocessors, except that the basic unit is a page instead of a cache line. Whenever a processor attempts to write to a page for which it does not have a local copy marked as writable, it communicates with the other processors to obtain a valid copy of the page (if necessary) and to invalidate all other copies in the system, before marking the local copy as writable. Whenever a processor attempts to read from a page that is not present locally, it again communicates with the other processors to obtain a read-only copy, possibly taking the write access permission away from another processor.

Another example of an algorithm that implements the shared data model was developed earlier by one of the authors\(^2\). In this algorithm, data is replicated at each site

\(^{1}\)as measured by lines of source code

\(^{2}\)A similar scheme is used by Bisiani and Forin [2].
so that all read operations can be serviced locally with no communication overhead. Write operations require that the data be multicast to all participating sites in an update packet. In order to provide for consistency, a central sequencing server is used to uniquely sequence all memory write operation on a system-wide basis. A write operation, therefore, entails the sending of the update to the sequencer, which appends a sequence number to the update before multicasting it to the other sites. By having the sequencer append sequence numbers that are gap-free, they can also be used in the implementation of a negative acknowledgment, reliable broadcast protocol, where a client can detect a missing packet through a missing sequence number (in which case it requests for a retransmission of the update packet). Hence, a write operation requires only two packets in the absence of failures. (Observe that any reliable, point-to-point message passing protocol also requires two packets, namely one for the message itself and one for the acknowledgment.)

One of the disadvantages of the shared data model is that the performance of the algorithms that implement this model depends highly on the access behavior of the client applications. For example, Li's SWMR algorithm performs well for applications with a high degree of locality in their memory accesses, where data within a page is not shared between processes at a fine granularity. In his thesis [9], Li shows that parallelized scientific applications using this shared data facility on a loosely-coupled cluster of workstations can exhibit excellent speedup, typical of a shared-memory multiprocessors. However, the performance of Li's algorithm can suffer from thrashing if the processes share data at a fine granularity within a page: If two processes alternate in access to data of a page, then the page will be migrated back and forth. The sequencing algorithm, on the other hand, performs well for applications that share data at a fine granularity if the ratio of read to write operations is high, but it does incur communication overhead for each individual write operation.

1.2 TORIS

In this paper we propose a new algorithm for implementing the shared data model; we call this algorithm TORIS. The most important characteristics of TORIS are:

Replication: Whenever a process accesses data that is not available locally, a replica of that data is made to the local site. Data which is replicated can be accessed by both read and write operations. In contrast, Li's algorithm replicates data for read-only access.

Optimism: With TORIS, read and write operations are performed on the local copy of shared data without ensuring that the replicas of shared data are consistent. TORIS is optimistic in that it determines a posteriori whether a process has accessed inconsistent data, in which case the process is rolled back to a previous consistent state. In contrast, pessimistic algorithms apply rules a priori to each shared data access to ensure that a process can access data only when the shared data space is and will remain consistent.

Transactions: In order to be able to roll back a process, TORIS logs accesses to shared data. To keep the size of these logs manageable and the operations on them efficient, data accesses are organized into transactions. Although transactions were introduced primarily to reduce the overhead of managing logs they also provide a mechanism for application-level process synchronization.

The TORIS algorithm evolved from attempts to increase performance of the sequencing algorithm described in Section 1.1. The basic idea was to coalesce multiple write operations into a single communication packet, thereby reducing both the aggregate cost of communication and the cost to sequence writes. Instead of obtaining a sequence number for each individual write operation, TORIS obtains a sequence number for a series of consecutive writes by a single processor. Obviously, this may lead to inconsistent copies of data, since writes are made to the local copy and only later transmitted (in batch) to other sites. If no processor accesses data while it is inconsistent, then temporary inconsistencies will not matter. Otherwise a conflict has occurred, in which case one of the processors will need to roll back to a consistent state. Hence, TORIS will perform well if the concurrently executing processes of an application do not access the same data at the same time in an inconsistent way.

In the next section we present the basic TORIS algorithm and justify its basic design. Section 3 considers a number of design issues in more detail. In Section 4 we analyze the performance of TORIS and show that it compares favorably to other algorithms that implement the shared data model for a class of applications. Finally, in Section 5, we consider a number of extensions to the basic TORIS algorithm that would further improve its performance.

For the purpose of this discussion, we assume a hardware base consisting of a cluster of workstations connected by a local area network. We also assume that all shared data is accessed through explicit read and write operations (although we realize that this may not be necessary with suitable compiler support or if the shared data facility were integrated with the local virtual memory system).

2 The Basic Algorithm

A TORIS transaction consists of any number of read and write operations bracketed by a begin_transaction and end_transaction, as depicted in Figure 1. (Other computation or access to local, non-shared data may occur within the transaction, but they are not considered part of the transaction.) The basic property of a transaction is atomicity, meaning that a transaction appears as a single indivisible (atomic) operation, even though it may be composed of multiple read and write operations.
begin-transaction
....
data ← read(address)
....
write(address, data)
....
status.tran ← end-transaction

Figure 1: A TORS Transaction

begin-transaction (begin-transaction)
write(address, data) (write)
read(address) (read)
status.tran ← end-transaction

Figure 2: TORS: Internal Structure

If a TORS transaction is aborted, all of the transaction's modifications to shared data are rolled back. If a transaction is committed, all of its modifications remain local to the site where the transaction is executed.

Transactions go through two phases, an active phase and a commitment phase. A transaction is in its active phase after executing begin-transaction and before executing end-transaction. While a transaction is active, TORS logs all read and write operations that are part of the transaction. A transaction enters its commitment phase when end-transaction is executed. At the beginning of the commitment phase a unique gap-free sequence number is assigned to the transaction by a central sequencer. This sequence number determines the order in which transactions are to be committed. The commitment phase ends when the transaction is either committed or aborted.

A transaction A with sequence number n is aborted if any concurrently executed transaction with a sequence number smaller than n has modified any of the data that transaction A accessed. This condition ensures that the transactions appear atomic.

The data structures used to implement the TORS algorithm are illustrated in Figure 2. Three logs hold the information necessary to detect that a transaction must abort, to roll back a transaction if it must abort, and to transmit the modifications performed by a transaction if it commits. The dependency log holds the address of the data item accessed by each read and write operation. The addresses in this log are compared with incoming modifications to detect if a transaction must abort. The rollback log holds for each write operation the address and old value of the modified data item. When a transaction is aborted, the previous value of modified data is restored from this log. The modification log holds the address and new value of the data item modified by each write operation. When a transaction commits, the modification log is sent to the remote sites so that they can update their replicas of shared data. While a local transaction is active, the modifications of committed transactions sent by remote sites are queued in the Remote Modification Queue and are not acted on until the transaction enters its commitment phase.

The begin-transaction operation creates the three logs. Read and write operations update the appropriate logs. The end-transaction operation performs the following:

1. Obtain the next gap-free sequence number from the global sequencer.
2. Wait for the modifications from all transactions with earlier sequence numbers. Determine whether inconsistent access occurred, in which case the local transaction is aborted. Apply each modification to the local replica.
3. Commit the transaction if it did not abort in Step 2. The modification log of the committed transaction is multicast to all other replica sites.
4. Discard the modification, dependency and rollback logs.3

Since the sequence numbers which are assigned to transactions are gap-free, they can be used in the implementation of a negative acknowledgment, reliable broadcast protocol, in the same manner as the sequencing algorithm described in Section 1.1. Hence, the transmission of the modification log of a committed transaction requires only three packets in the absence of failures, two packets to acquire a sequence number from the global sequencer and one packet to multicast the modification log.

TORS only logs accesses to shared data. Hence, only shared data can be restored if a transaction aborts. If a transaction aborts, TORS returns the appropriate status information from the call to end-transaction. The application is responsible for restoring local data modified by an aborted transaction.

Discussion of the TORS Algorithm

The three most important characteristics of TORS in order of importance are: replication of data to all sites where the

3Note, in order to allow retransmission of messages, the modification log is maintained for a period of time.
data is accessed, the optimistic algorithm, and transactions. These characteristics are considered in more detail in the following discussion.

With TORIS, data is replicated for both read and write access. We call this sort of replication Multiple Writer Multiple Reader (MWMR) replication. Replication has the advantage that data accesses can be serviced at the local site. Some algorithms, which do not use replication, statically assign shared data to particular servers, which are responsible for servicing accesses to the data they manage. These algorithms can be costly because of the communication incurred by each data access.

Two alternatives to MWMR replication which do not incur communication costs on each access are:

- **Migration**, where an access to data causes the data to be moved to the site of the access.
- **Single Writer Multiple Reader (SWMR)** replication, where data is replicated for read only access. Li’s Dynamic Distributed Manager algorithm [10] is an example of such an algorithm.

These alternatives have the advantage that when processes access data with a high locality of reference\(^4\) and seldom access the same data other processes are accessing, communication occurs infrequently. On the other hand it is also possible for thrashing to occur, where pages are copied back and forth between different sites several times in short succession. With MWMR replication thrashing does not occur, since data is replicated at all sites where it is accessed. However, a certain amount of communication is always necessary, since a write at one site must be sent to all other sites where that data is replicated. We chose to use MWMR replication in our algorithm because of its more stable behavior without thrashing.

The main motivation for using an optimistic algorithm is to reduce the amount of communication that is necessary. The TORIS algorithm has the advantage that the cost of communication is amortized over multiple write operations. A pessimistic algorithm ensures a priori that data is accessed only when it is consistent. Previously proposed, pessimistic algorithms for fully replicated shared data either incur communication costs on every write operation\(^5\) or communicate both before and after a sequence of write operations because they use some form of locking to ensure consistency. With TORIS, communication occurs only after a sequence of accesses.

Disadvantages of using an optimistic algorithm are: 1) the overhead to maintain logs and detect conflicts and 2) the cost associated with rolling back processes. However, in a loosely coupled distributed system, the costs associated with communication are very large. Therefore, if TORIS involves less communication then other algorithms which implement fully replicated shared data it is likely to perform better than these other algorithms.

Three decisions are crucial for the efficient operation of an optimistic algorithm that records information in logs:

1. The time at which checkpointing is done.\(^6\)
2. The time at which modifications are transmitted to other sites.
3. The time at which the logs can be discarded.

Transactions are a natural way to address these issues. In TORIS, checkpointing is done at the beginning of a transaction, when the process state is known to be consistent. Modifications are transmitted only after a transaction has committed. All records of accessed data (i.e. logs) can be discarded after the transaction has committed.

It is interesting to consider several alternatives to the approach used by TORIS. One alternative is to checkpoint the state of a process several times between the transmission of modifications. In this case, a process may not have to roll back to the beginning of the transaction. However, since the application is responsible for restoring its own local state, it becomes more difficult to coordinate the checkpoints of the application with the checkpoints within TORIS.

Another alternative is to transmit modifications several times between checkpoints. The difficulty with this approach is that it can result in cascaded roll back, since processes may access data modified by other processes which have to roll back. Since an application can roll back arbitrarily far, the logs can never be discarded.

Although transactions were introduced in TORIS for efficiently managing logs, they can also be used by applications for synchronization purposes. Since transactions are used to implement the shared data model, synchronization using transactions incurs no additional overhead.

Another reason for using transactions is that they have advantages from a fault tolerance point of view. Shared data is never modified by a transaction that does not complete successfully, simplifying the error recovery. Moreover, transactions can also be used naturally to develop fault tolerant applications.

### 3 Design Issues

In this section, we discuss a number of design issues of TORIS. A full discussion of the design issues which affect TORIS can be found in [6].

#### 3.1 Sequencing Transactions on Completion

The algorithm for end.transaction discussed in Section 2 was simplified in order to make it easier to understand. In the actual implementation, 'TORIS applies the queued'.
remote modifications before requesting a sequence number. If a conflict is detected before a sequence number is requested then the local transaction can be aborted without requesting and multicasting the transaction's sequence number to all other sites. Hence, by assigning sequence numbers only after a transaction has completed its active phase some abortions do not become visible to other sites.

Another advantage in assigning sequence numbers after transactions have completed their active phase is that it reduces the amount of time transactions must wait for other transactions to complete. For example, consider the case of a long transaction (i.e., a transaction which takes a long time to execute) and a short transaction executed concurrently. If sequence numbers were assigned to transactions when they are created, and the long transaction is assigned an early sequence number, then the small transaction (with a later sequence number) must wait for the long transaction to commit before it can be committed.

An advantage in assigning sequence numbers at the end of the active phase of a transaction is that a transaction must always wait for a sequence number before it can commit. When only one transaction executes at a given time, then assigning a sequence number early in the transaction will give the sequencer time to respond while the transaction is executing, reducing the time taken to commit the transaction.

### 3.2 Committing Transactions at the Local Site

In TORIS, the decision whether to abort or commit a transaction is taken at the site where that transaction is executed. An alternative approach, common in database systems[1], is to send the transaction's modification and dependency logs to a central server, which determines whether the transaction should commit or abort. A sequence number is assigned only if the transaction commits.

This centralized approach has the advantage that the central server can assign sequence numbers in such a way as to minimize the number of transactions that must abort. For example, consider the following two transactions:

\[
T_1 : \text{read}(A), \text{write}(A) \\
T_2 : \text{read}(A), \text{write}(B)
\]

If \(T_1\) and \(T_2\) are executed concurrently it could happen that the TORIS sequencer assigns \(T_1\) a smaller sequence number than \(T_2\), which would force transaction \(T_2\) to abort. In contrast, if a central server is used, transaction \(T_2\) could be assigned an earlier sequence number so that both transactions succeed. Note, however, that the central server will need to delay the commitment of some transactions in order to allow it to determine the optimal assignment of sequence numbers. Delaying commitment of transactions in database systems is acceptable, since the site that executed the transaction will usually be able to execute another independent transaction. However, for implementing shared data this approach is not acceptable, since the site that executed the transaction is blocked until the transaction commits. Moreover, if transactions are delayed, then it becomes more probable that the replicas of shared data will become more inconsistent, increasing the chance that other transactions will have to abort.

There are three advantages of the distributed approach (used by TORIS), where a transaction is committed at the site where it is executed. First, the amount of computation required to detect a conflict in the distributed approach is smaller than that needed in the centralized approach. Second, the work of detecting conflicts is distributed over multiple sites, reducing contention at the sequencer. Third, the distributed approach never requires the dependency log to be transmitted to other sites. Since, for most applications, reads are more common than writes, one would expect the distributed approach to require smaller messages than the central server approach.

### 3.3 Queueing Modifications rather than Applying them to the Local Replica as soon as they arrive

When a local transaction is in progress, arriving modifications are queued and only applied to the local replica during the end transaction routine. Modifications could also be applied as soon as they arrive (using a concurrent process), resulting in a number of advantages. First, it reduces the probability of a conflict, since the local transaction will not have to abort if a remote modification is applied before the data is accessed by the local transaction. Second, applying modifications as soon as they arrive at a site allows conflicts to be detected earlier in the execution of a transaction, forcing a transaction to abort early and reducing the amount of time spent on useless computation. Finally, the computation costs of detecting conflicts can be reduced since modifications are compared to dependency logs which, on average, are shorter than at the end of the transaction.

On the other hand, applying remote modifications early makes it necessary to access the shared data and dependency logs within a critical section to guarantee consistency. Since data accesses are the most common operation in TORIS, and since each access to data requires the logs to be updated, we decided to process remote modifications at the end of an ongoing transaction. However, reservations exist with respect to this decision and it is still under study. (Note that remote modifications are applied immediately if there is no active local transaction.)

### 4 Performance

In this section we present the results of a simulation study to compare the effect of locality of reference on the performance of TORIS and other algorithms for implementing the shared data model. The effect of transaction length on the performance of TORIS is also examined. The algorithms which we compare against are Li's Dynamic Distributed Manager (DDM) algorithm[10] and a migration algorithm.

One problem with simulating the shared data model
research has been done to characterize the way parallel applications access data. For this reason, we do not believe that a model of data access behavior is required. Little simulations can be used to accurately predict performance with respect to different data access characteristics. Namely to determine the sensitivity of different algorithms to data access distributions. The particular distribution chosen is a normal probability distribution, dependent on recent accesses by the same process. The normal distribution with a constant rate of access to shared data. In the case of TORIS this is equal to the rate of access to shared data resulting from transactions which do not abort. Test 1, considers the effect of locality of reference on the different algorithms. Test 2 looks at the effect of transaction size on the performance of TORIS.

The performance metric used in our analysis is Rate Successful Accesses. In the case of the migration and DDM algorithms this is equal to the average rate of access to shared data. In the case of TORIS this is equal to the rate of access to shared data resulting from transactions which do not abort. Test 1, considers the effect of locality of reference on the different algorithms. Test 2 looks at the effect of transaction size on the performance of TORIS.

The results obtained from the simulations and analytical models were consistent, therefore, we will show only simulation results in this paper. A more detailed discussion of the simulation and analytical results can be found in [6].

Table 1: Parameters for Different Tests

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Test Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of processors</td>
<td>1</td>
</tr>
<tr>
<td>cost packet event (μSec)</td>
<td>5</td>
</tr>
<tr>
<td>page size (bytes)</td>
<td>1000</td>
</tr>
<tr>
<td>large packet size (μSec)</td>
<td>256</td>
</tr>
<tr>
<td>amount of shared data ×1024 (words)</td>
<td>100000</td>
</tr>
<tr>
<td>read to write ratio</td>
<td>10</td>
</tr>
<tr>
<td>average time between access (μSec)</td>
<td>10</td>
</tr>
<tr>
<td>transaction length (words)</td>
<td>1000</td>
</tr>
<tr>
<td>variance x64 (words)</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 3: Test 1 - Different Locality of Reference

input parameters for the tests shown below.

The Migration and DDM algorithms perform much better than TORIS when there is a high degree of locality of reference, since almost no communication is required. However, for a lower degree of locality of reference the DDM and migration algorithms start to thrash while TORIS has relatively consistent behavior over a large range of locality of reference.

The reason TORIS has relatively stable performance is that performance is mainly determined by the amount of communication. With TORIS, most communication occurs when transactions are committed. Abortions are rare because conflicts are detected at a granularity of individual words. Moreover, most abortions that do occur tend to occur before a sequence number has been requested for the aborting transaction, in which case the abortion causes no communication.

The Migration and DDM algorithms on the other hand incur communication costs whenever a process accesses data which was previously accessed by another process, in which case an entire page is migrated (or copied) between sites. Therefore, with a low locality of reference there is a large probability that an access to shared data will require communication.

It is interesting to note that similar results are obtained when the read to write ratio, rate of access to shared data, or the amount of data being shared are varied. In general, TORIS displays relatively stable performance over a broad
range of data access behavior while the DDM and Migration algorithms perform better than TORS under extreme characteristics.

As one would expect, the performance of TORS depends on the length of transactions. Figure 4 shows the performance of TORS for transactions whose length varies from 10 to 10000 words. As the transaction size increases, the number of successful accesses reaches a plateau. This plateau occurs mainly because of the large packet cost which is a function of the number of bytes in a message. As the size of transactions increase, the number of modifications each transaction makes increases, and eventually communication costs are dominated by the large packet cost. Another reason for the plateau is that as transaction length increase the replicas tend to become more inconsistent and the probability of conflict increases.

In addition to the tests described in this section, a number of other simulations have been performed. For the sort of applications we considered, these tests indicate that TORS will perform well for up to ten processors. Beyond this, the centralized nature of the sequencer and the costs incurred at all sites where a multicast is received begin to limit performance.

TORS has also been compared to a sequencing algorithm, and a number of algorithms based on locks. For most of the conditions we examined, the best three algorithms were the DDM, migration, and TORS algorithms (although conditions could be defined in such a way that any of the algorithms performed better than the others). The advantage shared by the DDM, migration and TORS algorithms is that a number of modifications to shared data that is determined by the application and may vary from transaction to transaction.

5 Extensions

The cost of TORS can be primarily attributed to the three factors:

1. the cost to compare remote modifications to the dependency log at the sites where modifications are received,
2. the cost to receive modifications at the different sites, and
3. the time processes spend blocked waiting for transactions with earlier sequence numbers to commit.

In this section we briefly consider a number of the extensions to the basic TORS algorithm which attempt to reduce these costs. A more detailed discussion of these extensions can be found in [6].

5.1 Partitioning Data into Pages

As in any virtual memory system, it is possible to divide shared data into pages and to replicate only those pages that are currently being accessed at the site. Summary information of which pages a transaction has accessed can be maintained. In this case, the cost of detecting conflicts can be reduced by first coarsely comparing the remote modification with local access on a per page basis. Only in the case that both the remote modification and the local transaction access the same pages are further comparisons necessary. This cost reduction will be especially high if processes exhibit a high locality of reference and processes seldom access the same pages concurrently. We are currently studying various strategies for replacing and invalidating replicated pages. The hit rate of access to the shared data increases with the amount of data being replicated locally. However, the cost of detecting conflicts also increases with the size of the local replica.

5.2 Segmenting Data

In the basic TORS algorithm, modifications are multicast to all participating sites. The number of packets that must be received would be reduced with the attendant reduction in costs, if only those sites which need to receive a modification actually receive it. One approach to reduce the number of sites which receive a modification is to organize the shared data into segments, with each segment being managed by an independent sequencer. A transaction that modifies only a single segment need be sent to only those sites where a process is accessing that segment. Concurrent transactions must be assigned sequence numbers in the same order to all segments (i.e. if transaction \( \phi \) and \( \psi \) both access segments A and B, and transaction \( \phi \) has been assigned a lower sequence number than transaction \( \psi \) for segment A, then transaction \( \phi \) must also be assigned a lower
sequence number than transaction \( \psi \) for segment B) otherwise transaction atomicity would not be retained, and a deadlock condition could arise. It is possible to ensure that sequence numbers are assigned in such a manner, by organizing sequencers in a hierarchy (tree), where leaf sequencers are responsible for assigning sequence numbers, and higher level sequencers are responsible for ordering the requests of sequence numbers from the leaf sequencers.

### 5.3 Supporting Long Transactions

One difficulty with the basic TORS algorithm is that it does not handle long transactions (i.e., transactions that take a long time to execute) well. A long transaction will typically have a large read set. Moreover, since it has a long execution time, many other transactions may commit while it is executing. Hence, there is a high probability that a long transaction will abort due to conflicts with other transactions. Similarly, since a long transaction typically has a large write set, it may cause many other transactions to abort.

In order to reduce the impact of long transactions on performance, it would be possible to allow the application to specify:

1. when remote modifications should be applied to the local replica, and
2. when a sequence number for a transaction should be requested.

The former extension, gives the advantages of applying modifications before the completion of the local transaction (as described in Section 3.3), without the associated difficulties. The latter extension, allows the application to improve the chance that certain transactions will commit (possibly at the expense of other transactions).

### 5.4 Consistency Levels

In the basic algorithm, transactions are aborted whenever it is possible that data accessed was not consistent. However, not all applications require full consistency. For example, consider a simulated annealing algorithm [5]. During the initial stages of the algorithm, poor results are acceptable and a reduced level of consistency is tolerable. The results must be more exact during later stages of the algorithm, and, therefore, consistency becomes more important as the simulation progresses.

TORS allows applications to specify the consistency level at which each transaction is executed. The following levels of consistency are defined:

1. No reliability is enforced (i.e., some sites may not receive modifications and modifications may be received in different orders at different sites),
2. Modifications are reliably received and ordered,
3. Modifications are reliably received, are ordered, and transactions are rolled back:
   (a) if a remote modification modifies data modified by the transaction,
   (b) if a remote modification modifies data accessed by the local transaction (i.e., full consistency).

A transaction with a reduced level of consistency incurs far lower costs. For example, with consistency level 1, since there is no reliability or ordering constraint, no sequence number need be requested for a transaction. Moreover, an unreliable communication protocol can be used. Transactions executing at consistency levels 1 or 2 will never have to roll back. As a result, dependency logs do not have to be maintained, reducing the overhead incurred by the read and write operations. Since transactions will not roll back, a process executing a transaction is immediately returned a SUCCESSFUL status by end_transaction, allowing the process to proceed immediately with further local execution.

### 5.5 Asynchronous Transactions

The final extension to the basic algorithm, that we present here, allows a process to continue executing while its previous transactions are still in the commitment phase. Rather than blocking until a transaction commits or aborts, the process uses this time to execute both local computation and possibly additional transactions. With this extension, transactions commit and abort asynchronously. A variant on the idea of asynchronous transactions is to allow a process to start a new transaction before calling end_transaction for a current transaction. Applications that can use asynchronous transactions, take advantage of the time that would otherwise be spent blocked, waiting for transactions to commit, to execute useful computation.

### 6 Conclusions

In summary, we presented a new algorithm, TORS, for implementing the shared data model of interprocess communication for a distributed environment. The most interesting characteristics of TORS are:

1. Data is replicated at all processing sites.
2. An optimistic algorithm is used to:
   (a) maintain consistency between replicas of shared data, and
   (b) ensure that processes’ synchronization requirements are met.
3. Transactions are used to:
   (a) improve the performance of the optimistic algorithm, and
   (b) provide a mechanism for application level synchronization.

Assuming that the hardware alone provides for reasonable reliability.
Compared to other implementations of the shared data model, such as the sequencing algorithm, the migration algorithm and Li’s SWMR algorithm, TORiS has a number of advantages:

Performance: In distributed systems, performance is largely dictated by the amount of communication. TORiS reduces communication by using optimistic assumptions on consistency and combining together several write operations into a single communication packet. Li’s algorithm amortizes the cost of communication over several writes by moving a page to the processor where the writes occur. With Li’s algorithm, if the locality of reference is high, then performance is excellent, since communication seldom occurs. However, if locality of reference is low, the rate of migration becomes high, leading to thrashing behavior in the extreme case. In TORiS, if the optimistic assumptions are not correct, then transactions abort and have to be executed again, but, since communication stems mostly from the commitment of transactions, communication does not increase significantly in this case. Therefore, TORiS displays relatively stable performance over a much broader range of data access behavior.

Synchronization: We believe that TORiS is more efficient in controlling and synchronizing access to common data. Transactions are naturally used for this purpose — they are commonly used by database applications. Moreover, because transactions are used to implement the shared data model, synchronization using transactions incurs no additional overhead.

Fault Tolerance: TORiS applications structure their access to shared data into transactions. This has advantages from a fault tolerance point of view. Shared data is never modified by a transaction that does not complete successfully, simplifying the task of recovering from error. Moreover, transactions can also be used naturally to develop distributed fault tolerant applications.

The TORiS algorithm is similar to an optimistic algorithm proposed by Kung and Robinson [7] for database systems. However, the characteristics and requirements of database applications are very different from those of distributed applications, so the two algorithms differ substantially in detail. For example, applications using TORiS can access shared data at the fine granularity of individual words (or even bytes) while data base applications usually access data at a larger granularity. For this reason TORiS replicates shared data for both read and write operations. Furthermore, transactions of database applications are normally independent. Therefore, when a site is waiting for a transaction to commit it can usually execute another transaction. With TORiS, the transactions executed by a process will normally depend on the previous transactions executed by the same process. Therefore, in the common case, a process will have to wait for a transaction to commit before it can perform subsequent operations.

The current status of TORiS is as follows. We have implemented an initial prototype on a network of Sun workstations. We are currently going through a second iteration to fine tune and optimize the implementation. Further work is necessary to characterize the applications that run well under TORiS. For this, we need to implement and measure the performance of many more applications. Finally, we believe that an optimistic approach similar to that used in TORiS could be used for maintaining cache consistency for multiprocessors. We are currently looking into this.

References


