A DISTRIBUTED DATABASE SYSTEM USING OPTIMISTIC CONCURRENCY CONTROL

by

Dennis Heimbigner

CU-CS-281-84 August, 1984

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

A DISTRIBUTED DATABASE SYSTEM USING OPTIMISTIC CONCURRENCY CONTROL

by

Dennis Heimbigner

CU-CS-281-84 August, 1984

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION.

Submitted to the 5th
IEEE International
Conference on
Distributed Computing.

A Distributed Database System
Using Optimistic Concurrency Control

Dennis Heimbigner[1]
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

ABSTRACT

This paper presents an architecture for a distributed database. The principal feature of the database is its use of an optimistic concurrency control mechanism. An optimistic method assumes that transactions will complete and therefore the cost of constant locking is unnecessary. This system uses a version of the Kung and Robinson algorithm and extends it to the distributed case, but without the need to ship read and write sets. The database has been implemented, and is intended to serve as the basis for performance comparisons of various concurrency control schemes. The focus in this paper is on optimistic methods versus locking, and the relative merits of these two schemes are discussed.

[1] This research was supported by a University of Colorado Summer Research Initiation Foundation Fellowship (1984).

1. Introduction

Over the past few years, there have been some proposals for new methods of concurrency control: so-called "optimistic methods". An optimistic method assumes that very few transactions will ever interfere with each other, and so the overhead of dynamic locking may be unnecessary. This can be contrasted with locking, which assumes, pessimistically, that conflicts should be prevented by locking all data accesses.

The basic idea behind any optimistic method is to let a transaction run to completion and check for interference after the transaction is finished. If this checking is fast enough, and few transactions ever interfere, then the performance may be superior to locking methods.

In this paper we describe the architecture for a distributed database that uses optimistic concurrency control. The database system was designed for two rather distinct purposes. First, and the subject of this paper, it is to be used to discover the problems inherent in the implementation of an optimistic method in a distributed environment, and to explore the performance of various concurrency schemes, although the immediate emphasis is on optimistic methods. Second, it is intended to serve as the basis for a separate project into federated databases (see [Heimbigner 82, 81]),

In this paper, we first discuss the original optimistic method of Kung and Robinson and then extend it to handle distributed transactions. After that, we discuss the architecture of our database, and finally we discuss some issues in optimistic distributed databases. It is assumed that the reader has a knowledge of locking methods.

2. Optimistic Methods for Concurrency Control

Kung and Robinson [Kung 81] apparently were the first to use the term "optimistic concurrency control" and most later work stems from their paper. As an aside, Badal's work [Badal 79] amy also be mentioned since it is essentially an optimistic method and predates Kung and Robinson.

In Kung and Robinson's proposal, each transaction is divided into three phases: a read phase, a validation phase, and a write phase. In the read phase. a transaction proceeds to access (read, write, create, delete) named objects. As a transaction proceeds, the names of all of the objects that it reads are maintained in a read-set, and its modifications are maintained in a write set (actually three sets: one for modification, one for deletion, and one for creation). As for two-phase locking, any modification to an object, including creation and deletion, is kept local to the transaction so that no other transaction will see the modifications until the transaction completes successfully. If a transaction aborts before finishing its read phase, then its actions are thrown away.

Kung and Robinson are deliberately vague about the nature of the objects being controlled. For our purposes, an object is a record in a file. This record may be used to store a record of a relation (assuming a relational model) or a record of a btree used for indexing, or a record of a database schema.

When the transaction terminates its read phase successfully, it enters its validation phase. A transaction is validated by checking its read and write sets against those of any other transaction that could potentially interfere with it. If interference is found, then the validating transaction aborts by releasing its resources, and restarting.

The third phase is the write phase. If a transaction validates successfully, it proceeds to make permanent its modifications to the database. It does so by writing out its modifications. and returning its deletions to the list of free resources.

For the purposes of the algorithm, a read-only transaction is also validated to ensure that its answer is correct. thus a read-only transaction may possibly be invalidated and restarted.

The validation process is the key element of the optimistic method, and it hinges on detecting interference between two transactions. As an aside, we will mention that if we the read and write sets were augmented with some form of timestamps, certain special cases of non-interference could be detected that are currently treated as interference. But, barring that extension, and using two of the three rules of Kung and Robinson, two Transactions, T1 and T2, do not interfere if one of the following is true:

Condition 1:

T1 completes its write phase before T2 starts its read phase. In this case, T2 will never read a data item that is later invalidated by T1 (read-write conflict).

Condition 2:

The write set of T1 does not intersect the read set of T2, and T1 completes its write phase before T2 starts its write phase. The first clause guarantees that T1 will not invalidate any of T2's reads (read-write conflicts) The second clause guarantees that T2 will not invalidate any of T1's writes (write-write conflict).

Using these two conditions would require that the validation and write phase be carried out as an indivisible operation. If we are willing to pay some cost in additional transaction aborts, we can add a third condition that allows transactions to perform their validation and write phases in parallel to gain more concurrency:

Condition 3:

The write set of T1 does not intersect the read set or write set of T2 and T1 completes its read phase before T2 completes its read phase. Again, the first clause guarantees that T1's writes will not affect the outcome of T2.

When a transaction enters its validation phase, it is, in effect, being assigned its place in the equivalent serial order. Thus the order in which transactions enter validation will be the order in which they serialize, assuming that they do not abort. In the case of abort, they do not appear in the serial order at all.

Using this serial order, it should be clear that a validating transaction (call it Ti) need not check for interference with all transactions that have ever existed. Any transaction that has completed before Ti begins cannot possibly interfere with Ti (essentially condition 1 above). Any transaction that enters validation after Ti begins its validation phase is also irrelevant because it will appear later in the serialization. It will be the duty of that transaction to compare itself against Ti and possibly abort itself. Ti need not be concerned.

Thus, Ti must concern itself with those transactions that overlap it in time and have entered validation before Ti has entered validation. Again, these overlapping transactions may be divided into two sets. Those transactions that finish completely (i.e., through the end of the write phase) before Ti enters its validation can only affect Ti if they overwrote something that Ti read in its read phase. Thus, Ti need only intersect its read set with the write sets of such "finished" transactions. This is condition 2 from above. Kung and Robinson call this class of transactions the finish set.

The remaining transactions can interfere with Ti through conflicts with either the read set or the write set of Ti (condition 3). Ti must intersect its read and write sets against the write sets of these transactions. Kung and Robinson refer to these transactions as the active set.

In order to identify the various classes of transactions, a transaction number is assigned to each transaction. These numbers are assigned at the end of the write phase. If Ti saves the highest number (the StartTn) assigned just before it starts its read phase, then it knows that all transactions with numbers no greater match condition 1 and thus need not be checked at all. When it enters validation, it can again save the latest assigned transaction number (the FinishTn) and know that all transactions, Tj, from StartTn through FinishTn match condition 2, and so need only intersect Read(Ti) against Write(Tj) to check for interference.

It should be noted that the transaction numbers assigned in this way do not correspond to the serial order. Two transaction may enter validation in one order and yet leave the write phase in a different order if the first transaction to enter validation has a much longer write phase than the second transaction. If we want a number that corresponds to the serial order, then it must be assigned at the time that the transaction enters the validation phase.

3. Distributed Concurrency control

Extending transactions to the distributed database environment adds substantial complexity. A distributed transaction may actually consist of several local transactions (sometimes called cohorts) and an initiating transaction called the coordinator. In effect, the cohorts serve as surrogates for the original transaction, but they execute on the various component databases in a network [Gray 78].

Control (concurrency and recovery) is more difficult because local conditions do not necessarily guarantee global conditions. That is, even if each node obeys the rules used by a non-distributed system, the global effect may still lead to non-serializable behavior or inconsistencies. For example, two-phase locking with delayed writes may be used by each of the cohorts to handle the concurrency and recovery on a single machine. But it is possible for some cohorts to commit and some to abort and this can lead to inconsistencies in the database. If any of the cohorts aborts, then we want all cohorts to abort. If all cohorts reach the commit point, then we want to ensure that they actually commit, even in the face of the failure of some of the computers.

One algorithm for this, called two-phase commit, operates, briefly, by allowing all cohorts to commit if and only if each cohort validates successfully. If any cohort aborts, then all cohorts are required to abort. There are, of course, some details of acknowledgement, writing data to nonvolatile storage, and re-sending messages to handle processor failures. See [Gray 78] for details.

Distributed locking also has some problems. It is possible for deadlock cycles to span the boundaries of a machine. Thus, each machine may have a cycle free wait-for graph but when pieced together, the global graph may contain a cycle. Normally this is handled by passing around relevant pieces of the local graphs so that global checks can be made.

4. Distributed Optimistic Methods

As for other schemes, distribution of optimistic concurrency controls is not quite as simple as it seems at first. It too suffers from the problem that local consistency does not guarantee global consistency. To see this, consider the following execution on two computer nodes: N1 and N2, by two distributed transactions: T1 and T2.

N1	N2
T1 reads X	T2 reads Y
T2 writes X	T1 writes Y
T1 enters validation	T2 enters validation
T2 enters validation	T1 enters validation

At N1, T1 validates (there are no previous transactions) and T2 also validates because Read(T2) = Write(T1) = empty. By symmetry, at N2, T1 and T2 validate also. But, globally, there is no serial equivalent to this schedule. The order (T1,T2) fails because then the Y that T2 reads at N2 would be the one written by T1, but it isn't. Similarly, (T2,T1) fails because T1 would see the X written by T2, and it doesn't.

The difficulty here is that each node is picking a different serial order at each node, and the two orders conflict. There are a number of approaches to dealing with this problem. First, note that this can only occur when two transactions occur in each other's active set at different nodes. That is, each sees the other (at some node) as entering validation first. In those cases where we could know that the same relative order occurred on all nodes, there would be no problem. Our first step out of this dilemma involves the two-phase commit. A transaction can only get a transaction number if it passes the commit, and that can only succeed if all cohorts validate successfully. Thus, any transaction that "sees" that transaction number at any node must have entered validation after that transaction had completed its writes at all nodes. It is possible that a transaction will see another transaction as completed at one node, and still in the active set at another node due to timing delays in assigning transaction numbers, but this will not cause any problem except some wasted motion. Thus, We need concern ourselves only with those transactions that are in an active set set at some node.

The next step is to somehow forestall the conflicting serializations for the active transactions. It is clear that if this is to happen, each of the two conflicting transactions (T1 and T2 in the example above) must see each other in some active set at some node. We can either detect this case or we can try to avoid it. Detection requires passing the active sets around to check for this kind of conflict. Avoidance, which is our current solution, can be achieved at some cost by requiring no conflicts of any kind for transactions in the active set at each node.

Using the rules previously given, we would normally intersect the read set and write set of a validating transaction against the write sets of the transactions in the active set. Now, we must augment this to also intersect the write set of a validating transaction against the read set of active transactions. If this intersection is non-empty, then the validating transaction fails.

Why does this work? It works because it only allows read-read intersection of any two transactions that may be simultaneously active. This means that any serialization conflict is illusory; it could be re-ordered to avoid the conflict. This in turn means that we can define a consistent serial order for these transactions.

There is a price for this, of course, and it is a potentially higher rate of transaction failures. Transactions which might have succeeded before, may now fail because of the extra intersection condition. At the moment we do not know how expensive this solution is since it depends upon load conditions and transaction behavior. As compensation, we avoid shipping active sets around to all the nodes of the network, and this may produce a substantial savings in time.

5. General Architecture

There has been some discussion of using Optimistic methods for distributed databases (see [Bhargava 82]). But little progress seems to have been made towards actually constructing and testing such a database. Starting from a student project in February of 1984, we have constructed a prototypical version of such a distributed database.

The database system runs as a collection of processes on a network of Sun workstations under Unix 4.2 and using the interprocess facilities of that operating system.

Figure 1 shows the set of user processes involved in evaluating a transaction. The system is session based in that when a user first invokes the database, a collection of processes is established to handle all of the transactions for that user. When the user is finished, the processes terminate. The per-user processes are of three types:

(1) The *user* process contains either the application program or some kind of user-interface program. It is loaded with a library of code to talk to the database system proper.

(2) The *coordinator* process is the main database process. One exists for every user. It's responsibilities include security checking, transaction management (voting for two-phase commit), and query decomposition.

(3) One *cohort* exists on every node that a user accesses during the course of a session. The cohort is responsible for local evaluation of queries and for local management of transactions. This latter involves collecting read and write sets, and testing for conflicts.

In general, the user process, the coordinator and one of the cohorts all reside on one node, but this is not required by the design.

In addition to the per-user processes set, there is a per-node set of processes (figure 2). The *server* is a well known process at every node, and it is used to establish the per-user processes. Thus, when a user program wishes to start up the database, it sends a message to the server on its local node, and that server then creates a coordinator process. Later, a coordinator will ask the server on a node to create a cohort when it needs data from that node.

This server structure is imposed by the networking facilities of Unix 4.2. When the server creates a process, it also makes a two-way stream connection (a pipe, if you will) between the requesting process and the new process. This is indicated back in figure 1 by the edges between the processes.

In addition to the server, each node has a *ccs* (concurrency control system) process. The ccs effectively serves as a form of shared memory for all the transactions on a node. It contains the transaction number counter, and pointers to the read and write sets of transactions. The actual sets are contained in disk files and serve as another form of simple shared data.

Each process in this system is message (or event) driven. An event is assumed to be indivisible within the process. Each process has a top loop that is of this form:

```
Loop:  1. wait for a message from any connected process
       2. perform (indivisibly) the operation indicated
          by the message
```

3. goto Loop.

It is assumed that the operation may send any number of messages, but it may not (with some exceptions) wait for any message to arrive. Thus, long operations that involve sending and receiving many messages (such as the query evaluator) are broken into many small operations whose sequencing is controlled by the flow of messages.

The general architecture was designed to isolate the upper levels (data model) from the lower levels (concurrency control). This can be accomplished by providing a firm interface at the level of physical records from files; we call this interface the heap level. Changes above and below this can be carried out with little affect on each other. For example, the database system uses the relational model for representing data, and the relational algebra for queries. This is all constructed in terms of the heap level and so it would be feasible to convert to some alternate model. On the other side, we can (and will) be changing the concurrency mechanisms used by the heap level without affecting the data model.

6. Transaction Management

When a user begins a transaction, he invokes a StartTransaction operation. This operation is passed to the coordinator, where the start of a new transaction is recorded. The coordinator's principal data structure is a record of the current state of the transaction: *done*, *started*, *ended*, and *decided*. The *ended* state signals the end of the read phase and that all cohorts have entered validation. The *decided* state indicates that all transactions have voted whether to abort or commit. As part of the decision procedure, the coordinator receives and counts the votes from the cohorts.

The coordinator also notifies that cohorts when a transaction starts. Cohorts are actually created only as needed, so a cohort may start up after a transaction has already begun. Each new cohort is synchronized with the transaction by ensuring that its first message is a StartTransaction operation.

Once the transaction is started, the details of concurrency control are the responsibility of the cohorts. The database system at each cohort is set up such that all accesses to data and index files are channelled through a single module that is also responsible for the concurrency control. There are 8 operations that can occur at this level: *start*, *abort*, *validate*, *dowrites*, *readrecord*, *writerecord*, *deleterecord*, and *createrecord*.

The first four operations are specific to transaction management. *Start* begins a transaction by requesting the ccs to record its existence. The ccs responds with the StartTn number. The *abort* operation may occur at any time as a result of an explicit user command, or because of some kind of error. In addition, it may occur implicitly during the two-phase commit if some other cohort has voted to abort. The *validate* operation signals the end of the read phase and the start of the validation phase. The validation phase involves 3 steps: (1) requesting the FinishTn number from the ccs, (2) writing its read and write sets to files, and (2) doing the set intersections. We will defer discussion of this second step. The *writephase* operation involves writing the modifications to the actual database to make them permanent and then notifying the ccs to assign a final transaction number.

The four operations of readrecord, writerecord, deleterecord, and createrecord, are the normal file and record access functions provided by the heap interface mentioned previously. The higher levels of btree access and the relational algebra operations are implemented in terms of these four operations. There are four sets corresponding to these four operations. Each set stores the record numbers of the records referenced by the corresponding operations. These sets are actually associated with the particular file being referenced, so that each data and index file actually has instances of the four sets. The sets are stored as hash tables indexed by record number. This allows fast insertion and fast testing.

The write set is slightly different from the others since it must also reference the new value of the modified record. These modified records are stored in a special file and a pointer to that new record is kept in the writeset file. The create set does not need to do this because its new records are allocated directly to free slots in the data file and the normal record number can serve as the pointer to the new data.

At the end of the read phase, the transaction must validate itself against the appropriate sets from other transactions. To do this, the transaction must find out which transactions are in its finish set and which are in its active set. This information is obtained by querying the ccs for a list. There is one element in the list for each transaction against which some form of test is needed. Each element contains the status of the corresponding transactions (finished or active) and pointers to the read and write set files for that transaction. Once a transaction has this list, it then proceeds to perform the appropriate intersections. Intersection consists of reading the read set or write set from the appropriate file. Each record reference stored in that file is hashed into the local set to see if there is a match. Since this operation is read-only, any number of transactions may simultaneously perform validation tests.

The ccs maintains several linked list data structures to record the relevant transaction information. Whenever a new transaction is started, the ccs adds an entry to the list of started transactions, and records the StartTn for it. Similarly, a transaction is added to a list of validating transactions when it signals that it is entering validation. Also, a number reflecting serial order, and the FinishTn are stored at this time. A subset of the validate list is sent to the validating transaction to indicate which other transactions it must check. This subset is determined by scanning the validate list looking for transactions that fit into one of two categories:

(1) It has entered validation before the requesting transaction and is not yet through the write phase.

(2) It has finished the write phase and it did not finish before the the requesting transaction started. That is, is the transaction number of the finished transaction greater than the StartTn of the requesting transaction?

Notice that transactions never need to be assigned explicit identifiers (before finishing, anyway) because they are implicitly identified by their position and state in the validate and start lists.

As a final note, it is not necessary for the ccs to track all transactions forever. As soon as the transaction number for a finished transaction is less than the StartTn for all active transactions, then all data about that transaction may be purged and the space for its read and write sets may be recovered. The ccs keeps track of this by periodically scanning its tables to identify such useless transactions.

7. A Preliminary Comparison to Locking

It is useful to summarize the merits of both optimistic methods and locking to see what features are important to measure and to understand the trade-offs made by each method.

Locking has been implemented many times, and its merits and demerits are well known:

(1) Locking requires shared memory of some sort to hold the lock table for all transactions. Further, this memory should be fast since it will be accessed for every record access.

(2) Locking has a reduced level of multi-programming because some transactions will be waiting for others to release locks.

(3) Transaction aborts may occur before the transaction is finished and so reduce the amount of time spent in re-doing transactions.

(4) Periodically, a global graph must be constructed to detect deadlocks.

A similar list of merits and demerits for optimistic methods might include the following:

(1) Slow forms of shared memory (such as a shared process) may be adequate since the accesses to it are infrequent.

(2) The read and write sets may consume substantial quantities of space.

(3) Performing the intersections of the read and write sets may be substantial.

(4) Potentially, the degree of multiprogramming can be very high since transactions need not wait for each other during execution.

8. An Approach to Performance Measurement

Comparing the performance of various concurrency control mechanisms is difficult. The few performance comparisons that have been carried out have used simulations [Bhargava 82, Peinl 83], presumably because it is easier to construct a simple simulation than to actually modify an existing database. The reports of these simulations have been flawed because they do not seem to represent some of the most important features of optimistic methods. In particular, it is not clear that any of these implement the condition 3 that allows parallel validation and write phase. In [Bhargava 82] they are explicitly treated as critical sections, and that clearly will affect the outcome of the simulation. Nonetheless, it is encouraging, but by no means conclusive, that both of these simulations support the view that optimistic methods are competitive with locking.

The essential problem with simulations is that they must be validated against a real system. There are very detailed system interactions that may seriously affect the performance of a concurrency control mechanism that may have been overlooked in a simple simulation. For example, locking protocols require shared memory to operate. Yet, if one were using a standard Unix 4.2, that feature is not available. Instead, shared memory has to be simulated using either shared files, or a shared process (such as the ccs). Both of these mechanisms favor optimistic methods, which do not require the fine grain of interaction of locking. Thus, any simulation that did not account for the nature of the shared memory would incorrectly bias its results towards locking. We believe that definitive comparisons can be performed only on a real implementation of a distributed database, and that is one of the goals of this project.

Even given a real database, many problems remain. One problem concerns the test set against which comparisons are made. Unfortunately, we do not have access to an actively used database from which we can extract representative data and transactions. Rather, we must generate synthetic data distributions and workloads for the comparison. This will involve generating synthetic relations with suitable distributions for various attributes and generating transactions with specified reference patterns.

Finally, we must decide what parameters are to be measured. One gross measure is transactions completed per second. But one would like more detailed measurements to understand why one method is preferable to another. Some other parameters of interest will be:

(1) Percentage of aborted transactions. In addition, for locking, we would like to know how close to finished is the transaction when aborted.

(2) Reference rate to shared memory. Again, for locking, we would like to know the percentage of locks that require waiting, which will translate into a degree of multiprogramming.

(3) Read and write set sizes.

(4) Time spent in validation.

(5) Time gained by parallel validation and write phases.

There are other parameters, but we believe that these will give the key insights into the comparisons of locking and optimistic methods.

9. Status

At the time that this paper was written, all of the database except the concurrency control was tested and working. The optimistic concurrency control was being tested. We are beginning to construct the tools and data necessary for the performance measurement. The system is available to anyone interested enough to send a tape, with the usual caveats about documentation and reliability.

This system is missing a number of features that are important to a production system, but are not immediately needed for the performance measurements. First, the system is set up to handle transactions that abort during validation, but more general crash recovery, including a write-ahead log, is not implemented. Second, there is no provision for long transactions. It is known that optimistic methods do not handle these well because they may starve. Finally, the distributed database does not handle duplicate data. We would expect that adding this will require more research to modify optimistic methods to handle this, and it will certainly affect performance.

Our first comparison will be against locking, and specifically against the locking method used by INGRES [Stonebraker 76]. That method stores locks in files, and uses the implicit shared memory provided by the operating system disk buffers. We also intend to modify Unix to provide shared memory and then implement locking directly in terms of that.

10. Summary

In this paper, we have described an architecture for a distributed database that uses optimistic concurrency control. We have shown how to extend a previous algorithm to the distributed case in a way that reduces global data passing. We have also discussed the relative merits of locking and optimistic methods.

References

[Badal 79] Badal, D. Z., "Correctness of Concurrency control and Implications in Distributed Databases", *Proceedings of COMPSAC 79*, pages 588-593.

[Bhargava 82] Bhargava, B., "Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comparisons to Locking", *Proceedings of the IEEE International Conference on Distributed Computing Systems*, 18 - 22 October 1982, Miami, Florida, Pages 508 - 517.

[Gray 78] Gray, J. N., "Notes on Data Base Operating Systems", *Operating Systems: An Advanced Course*, Bayer, R., Graham, R.M., and Seegmuller, G. (eds.) Springer Verlag Lecture Notes in Computer Science Volume 60, 1978, Pages 393-481.

[Heimbigner 82] Heimbigner, D. M., *A Federated Architecture for Database Systems*, Ph. D. Thesis, University of Southern California, also available as Technical Report TR-114, August 1982, Computer Science Department, University of Southern California.

[Heimbigner 81] Heimbigner, D. M. and McLeod, D., *Federated Information Bases - A Preliminary Report*, *Infotech State of the Art Reports*, Volume 9, Pergamon Infotech Limited, Maidenhead, U. K., 1981, Pages 383-410, M. Atkinson, ed.

[Kung 81] Kung, H. T. and Robinson, J. T., "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems* 6(2):213-226 (June 1981).

[Peinl 83] Peinl, P. and Reuter, A., "Empirical Comparison of Database Concurrency Control Schemes", *Ninth International Conference on Very Large Data Bases*, 31 October - 2 November 1983, Florence Italy, Pages 97 - 108.

[Stonebraker 76] Stonebraker, M., Wog, Eugene, Kreps, P., and Held, Gerald., "The Design and Implementation of INGRES", ACM Transactions on Database Systems 1(3): 189-222 (September 1976).

Figure 1. The Per-User Process Structure

Figure 2. The Per-Node Process Structure