# AN INVESTIGATION OF TYPE-SPECIFIC OPTIMISTIC, PESSIMISTIC, AND HYBRID CONCURRENCY CONTROL

by

**Yvonne Coady**

B.Sc., Gonzaga University, Spokane, 1985

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science



SIMON FRASER UNIVERSITY

December 1988

**APPROVAL**

Name: Yvonne Coady

Degree: Master of Science

Title of thesis: An Investigation of Type-Specific
Optimistic, Pessimistic, and Hybrid
Concurrency Control

Examining Committee:

Chairman: Dr. Joseph Peters

Dr. Stella Atkins
Senior Supervisor

Dr. Wo-Shun Luk
Committee Member

Dr. Tiko Kameda
External Examiner

Date Approved: December 9, 1988

Title of Thesis/Project/Extended Essay

An Investigation of Type-Specific Optimistic, Pessimistic, and Hybrid Concurrency Control

Author:

(signature)

Monica Yvonne Coady

(name)

December 9, 1988

(date)

## Abstract

In general-purpose distributed systems, abstract objects may be manipulated by concurrent actions called "extended transactions". A server provides the necessary access operations and concurrency control for each object. By exploiting semantic information, the standard pessimistic servers can be replaced with efficient optimistic servers, and also with hybrid servers combining features of both pessimistic and optimistic concurrency control. The finest granularity at which different synchronization methods can be used is on a per-conflict level.

This thesis describes a novel implementation of a hybrid server for a user-defined abstract object, a semiqueue. The semiqueue server may be customized so that every conflicting operation (executed by concurrent extended transactions) may be treated either pessimistically or optimistically, depending on the expected level of conflict. Performance studies show that this hybrid server is more efficient over a range of conflicts than a purely pessimistic or purely optimistic server when some types of conflicts are expected to be frequent, and others are expected to be rare. The results generalize to other shared data structures, showing the practicality and effectiveness of this approach.

## Acknowledgements

I would like to extend my gratitude and deep appreciation to Dr. Stella Atkins, *supervisor-extraordinaire*, for her endless supply of guidance, wisdom and support. I am also very grateful to my brother, Michael, whose relentless determination and enthusiasm will always be my greatest source of inspiration.

# Table of Contents

# List of Figures

# 1 Background and Fundamental Concepts

## 1.1 Introduction to Extended Transactions

Recently, some innovative research has been devoted to general purpose methodologies involving an extension of the traditional transaction model[1] which could potentially simplify the construction of general purpose distributed applications [Schwarz84, Spector83, Spector85, Weihl85, Herlihy86]. This renovation of the classical model provides distributed application programmers with a mechanism for customizing concurrency constraints on shared abstract objects and ensures an atomic (all-or-nothing) effect of a group of type-specific operations. As opposed to the traditional model, this *extended transaction model* applies to data structures other than records, operations other than simple READs and WRITEs, and applications outside of the conventional database domain. Consequently, *extended transactions* have the potential to be a valuable tool for organizing and structuring computations in general purpose distributed systems.

The purpose of extending the conventional transaction model to applications outside of the database domain is to simplify the construction of many types of

[1] A transaction typically consists of several READ operations performed on a set of records known as the transaction's *read set* , and/or several WRITE operations performed on a set of records known as the transaction's *write set* .

distributed programs. This can be accomplished by lightening the burden on application programmers through the simplification of synchronization issues. Further, the extended transaction model can provide a flexible way of maintaining arbitrary application-dependent consistency constraints without unnecessarily restricting concurrent processing of application requests.

In order to accommodate the additional requirements of general-purpose distributed systems, the extended transaction model must include an effective method of *concurrency control* for extended transactions operating on abstract data types. A concurrency control scheme allows operations from simultaneously active transactions to be interleaved in such a way that each transaction is provided with a consistent view of the system state, as if it were executing alone. This effect is formally known as *serializability* .

Frequently, the serializability requirement is too strong and can significantly reduce the amount of concurrency a synchronization mechanism has the potential to provide [Spector83]. For example, a *semiqueue* [Weihl85, Schwarz84] is a species of queue that allows for more concurrency than a strictly FIFO queue because it does not guarantee to dequeue items in the order they were enqueued, and as a result, all previously enqueued objects (that are not already locked) are eligible for dequeuing. Concurrency constraints for the nonserializable abstract type *semiqueue* are such that two *Enqueue* operations do not conflict, nor do an *Enqueue* and a *Dequeue* that access different items, and likewise, two *Dequeue* operations that attempt to remove different items do not conflict.

Exploitation of semantic information associated with the consistency constraints of type specific operations can also be used to achieve greater concurrency since well-informed decisions can be made regarding the necessity of transaction delay or abort. For example, consider the abstract type *directory* and a corresponding operation *Insert(entry)* [Spector83]. Within the context of simple READ and WRITE operations, *Insert* modifies the directory and therefore is considered to be equivalent to a WRITE operation. In terms of conventional database synchronization schemes, WRITE operations on the same data item from two concurrently active transactions always cause a conflict. Intuitively, however, when one considers the semantics of insertion, it is apparent that a conflict between concurrent transactions inserting different entries is nonessential. We return to this example later (in Section 6), showing how semantic information is used to reduce the number of conflicting operations to just a few.

## 1.2 Pessimistic and Optimistic Concurrency Control

Fundamental approaches used by concurrency control schemes can be broadly categorized as either *pessimistic* or *optimistic* methods. In a pessimistic approach, conflicts are identified during a transaction's execution and resolved by imposing a delay on some transactions. Conversely, in an optimistic approach, conflicts are identified at the end of a transaction's execution and resolved by aborting and

restarting some transactions at a later time. Standard pessimistic and optimistic concurrency control mechanisms for database management systems have been based on *two-phase locking* [Eswaran76] and *validation* [Kung81], respectively. The following sections give a high level description of the methods by which these two schemes achieve concurrency control.

### 1.2.1 Pessimistic Concurrency Control: Two-Phase Locking

In order for a locking technique to guarentee serializability, a transaction must proceed through two phases: a *growing* phase in which it must request all locks, and a subsequent *shrinking* phase wherein it releases locks and cannot issue any new lock requests [Eswaran76]. The rules associated with the allocation and releasing of locks on individual items are the following:

Rule 1. Seperate transactions cannot simultaneously possess locks which are in conflict.

Rule 2. When a transaction releases a lock it cannot obtain further locks.

Within a database environment where operations consist solely of READs and WRITEs, two-phase locking provides concurrency control through the use of READ-locks and WRITE-locks. These locks conflict in the following manner:

| | READ-lock(*item*) | WRITE-lock(*item*) |
|---|---|---|
| READ-lock(*item*) | | X |
| WRITE-lock(*item*) | X | X |

Hence, mutiple readers are allowed, but a readers and writers exclude each other, and a writer will also exclude other writers.

Although adherence to these rules guarentees serializability, one snag inherent in two-phase locking is *deadlock*, which may result whenever transactions are forced to wait for the release of certain locks (according to Rule 1 above). Consequently, some effective method of transaction abort must be included with this technique. In order to facilitate this, it is necessary for a transaction to hold all locks until the end of its execution, just in case it becomes necessary to undo its changes [Kung81]. This way, backing up an active transaction which has become part of a deadlock situation consists of undoing all of the effects of its updates, usually recorded on a log kept by the system, releasing all of its locks and restarting its execution.

### 1.2.2 Optimistic Concurrency Control: Validation

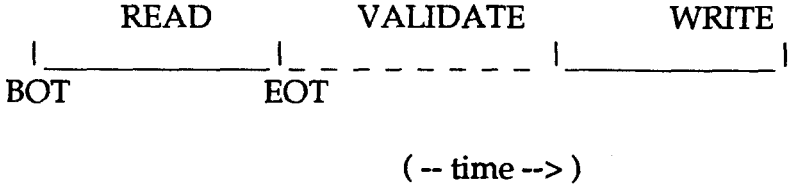
Originally, optimistic techniques for concurrency control in database systems totally eliminated locking due to the fact that the following undesirable properties are inherent in such schemes [Kung81]:

- The effective level of concurrency can be severely restricted since all locks are held until the end of a transaction.

- The overhead imposed by lock maintainence and deadlock detection can be substantial (e.g., 10% of total execution in System R [Harder83]) and unnecessary in some cases (e.g., READ-only transactions).

Consequently, given a situation where conflicts can be assumed to be rare, a locking mechanism is not only nonessential, but also expensive.

Within an optimistic approach, the transactions of a database can be considered to pass through at least two of the following three distinct phases [Harder83]:




The *READ* phase constitutes the "execution" of the transaction, hence all of the transaction's READ and/or WRITE operations are performed during this phase. An *intentions list* for each transaction is maintained such that operations performed on behalf of a given transaction are directed to this private buffer. Modifications to data are all inflicted upon local copies that remain in this buffer, as opposed to directly modifying the global item. The subsequent *VALIDATION* phase is entered

when a transaction has completed all its operations. This phase ensures that all the accesses and changes made by a transaction do not result in any loss of consistency within the database (i.e., it ensures serializability). Consequently, any violation of consistency involving the validating transaction will be detected as a conflict in this phase. If a conflict is uncovered during validation, it can be resolved by backing up the validating transaction and restarting it as new. This is accomplished simply by throwing out that transaction's intentions list and starting again. Finally, in the *WRITE* phase, all local copies of data in a transaction's intentions list are made global. Only those transactions that successfully complete validation enter into this phase, and the combined execution of the *VALIDATION* and subsequent *WRITE* phases of a transaction must proceed in an uninterrupted (i.e., atomic) fashion.

### 1.2.3 Integration of Pessimistic and Optimistic Techniques

An optimistic approach to concurrency control allows for the unsynchronized execution of transactions and relies on commit-time validation to detect conflicts and enforce serializability. As opposed to pessimistic techniques which introduce delays, within an optimistic scheme conflict resolution is accomplished by transaction abort. The element of optimisim in this approach stems from the fact that this technique is efficient only if validation succeeds with sufficient frequency. That is, an optimistic approach is only cost-effective if the level of conflict is

sufficiently low. Conversely, a robust pessimistic scheme is less efficient when the level of conflict is low (due to unnecessary locking overhead), and increasingly cost-effective when the conflict level is sufficiently high. As a result, optimistic and pessimistic approaches to concurrency control are appropriate under opposing sets of conditions, which can be characterized by the probability of conflict among transactions. This would seem to indicate that a cost-effective synchronization model should therefore be equipped with not just one technique or the other, but both [Kung81, Vidyasankar84, Herlihy86]. This dual mechanism, or hybrid approach, could then allow for the selective application of the most appropriate method of concurrency control on a situation dependent basis.

## 1.3 Thesis Overview

The work presented here is an exploration of a recently proposed method for synchronizing extended transactions accessing shared abstract objects through a server [Herlihy86]. This type-specific method supports a hybrid approach by the selective use of an optimistic concurrency control technique on a per conflict-type basis in conjunction with a two phase locking (pessimistic) approach. A purely optimistic, a purely pessimistic, and a hybrid server are designed according to this method and implemented for the abstract data type *semiqueue*. Through the evaluation of the performance and implementation issues associated with each

server, along with a further example of its application to the *directory* data type, we demonstrate the practicality and efficiency of this highly flexible method and establish the effectiveness of its hybrid support. Fundamental characteristics of type-specific optimistic, pessimistic and hybrid control are established, along with the identification of the most appropriate environments for each.

Section 2 gives an overview of previous work using semantic information to increase concurrency for extended transactions. Section 3 describes an example, the semiqueue data structure, the operations on the semiqueue, and the conflicts which arise between extended transactions accessing the object concurrently. Section 4 presents the implementation details of our hybrid semiqueue server, including a description of its data structures and the operations which use them. Section 5 presents the results of our tests, which subjected each of our servers to varying degrees of transaction conflict. This is followed in Section 6 by an evaluation of the respective behavior of each approach. To extend these results to other data structures, we then analyze the application of this method to a second data structure, a *directory* . Section 7 presents our conclusions and discusses possible directions for further research.

# 2 Previous Work on Concurrency Control for Extended Transactions

## 2.1 Fast Path: A Successful Approach for Database Transactions

A leading commercial database management system currently employed world-wide is IBM's IMS/VS [Gawlick85, O'Neil86]. The concurrency control scheme used by this system, known as *Fast Path* , has the proven ability to sustain impressive transaction rates within banking environments (180 Debit/Credit transactions per second reported in 1983 [Gawlick85]). One of the most innovative and successful features of this system's design (1974) is the incorporation of semantic information and optimistic techniques into its treatment of frequently accessed fields or "hot spot" data.

When applied to frequently accessed data such as shared counters, common locking techniques impose severe bottlenecks that can have a crippling effect on an entire database system. This result is inherent in the fact that once an item is locked by a transaction, it must remain locked for the duration of that transaction. Fast Path attempts to avoid these bottlenecks by abandoning conventional pessimisitic methods in "hot spot" situations. These situations most commonly involve *summary data* such as "Quantity On Hand" or "Total Cash Received", and are normally kept in main storage. Semantic information associated with these specific data types reveals that most of the updates involving these fields are of an

increment or decrement nature. Consequently, these operations can be most effciently made without explicit retrieval and replacement of the records. If, for example, the desired modification was to replace a quantity of 1000 by 1005, the database system can simply add 5 to 1000 [Gawlick85]. Operations on summary data are performed by MODIFY requests, that have the following form:

MODIFY: *database*
*record*
*attribute*
*operation* ( one of +, -, *, /, or :=)
*value*

Before actually modifying an attribute, however, Fast Path allows the programmer to issue special VERIFY requests to ensure that an attribute bears some relation (<, <=, =, >=, >) to a known value. These requests are checked again during commit processing. In keeping with the characteristic properties of optimistic techniques, the MODIFY requests are not actually performed until commit time, after all the VERIFY requests have been reevaluated. A VERIFY request has the following form:

VERIFY: *database*
*record*
*attribute*
*relation* ( one of <, <=, =, >=. >)
*value*

Due to the fact that an attribute is not actually updated until the end of a transaction, the first set of VERIFY requests are not crucial and are intended only to supply confidence since they cannot guarantee success. They can, however, provide the application program with a basis for executing an alternative branch of logic in the event that the verification criterion are not currently met. For example, if "Quantity On Hand" must remain nonnegative, the application could include:

```
if VERIFY qoh >= const ->
   MODIFY qoh:= qoh - const
else <alternative branch >
```

The second execution of the VERIFY requests are generated by the system at commit time and used to determine if a transaction's MODIFY requests should be executed and subsequently committed. In the case where verification fails, the transaction is aborted. Otherwise, the attribute is locked and subsequently modified. By postponing the actual modification of an attribute until the end of a transaction, Fast Path shortens the duration for which a lock is held on a "hot spot", and consequently is better able to avoid bottlenecks than common locking techniques.

## 2.2 Argus and TABS : Conventional Pessimistic Approaches for Extended Transactions

Argus [Liskov83] and TABS [Spector85] are two existing systems that provide pessimistic based support for extended transactions on user-defined abstract types.

Both rely on two phase locking and the standard definitions of READ/WRITE lock conflict. Argus is an integrated programming language/system designed to support the construction and execution of distributed programs, and the TABS prototype is a general purpose facility explicitly designed for extended transactions. Within both of these systems, a distributed program consists of a group of servers (known as *guardians* in Argus and*data servers* in TABS) communicating via operation invocations. Each server encapsulates one or more data objects and the operations used to manipulate them. These operations provide the sole means by which servers can access each other's data objects. Arguments are consequently passed by value in order to guarantee that all direct references to an object are contained within that object's server; processes within a single server, however, can share objects directly. Seperate processes are spawned to execute each operation invocation and each server must provide access control and synchronization for the data objects it contains.

Although both systems employ this client/server model to support transactions on shared abstract types, their subsequent user interfaces are very different, with Argus being the simpler of the two to use [Spector85]. To guarantee proper synchronization of concurrent transactions, Argus supplies a special MUTEX object to provide concurrency control and TABS relies on the semantics associated with coroutines and the application programmer to explicitly lock shared types. The following sections investigate the treatment of the abstract data type *semiqueue*

within Argus (section 2.2.1) and TABS (section 2.2.2), then briefly outline the TABS system and define the focus of our work within such a facility (section 2.2.2.1).

### 2.2.1 Argus

Argus provides linguistic support for *atomic actions* (i.e., transactions). Actions are the prinicipal way of performing computations in Argus -- they start at one guardian and can spread to others via operations invocations known as *handler* calls. Atomicity is accomplished through an *atomic data type* [Weihl85] construct which is composed of a set of operations and data objects that are implemented in such a way that concurrent actions accessing these objects are serializable. Some of these atomic data types are built-in (e.g., atomic arrays) and others can be user-defined.

Operations on built-in atomic data types are all classifed in terms of conventional READs and WRITEs and serializability is accomplished by an augmented two phase locking scheme. As with conventional two phase locking mechanisms, locks are acquired automatically when an action calls an operation and are held until that action commits or aborts. In Argus, however, when a write lock is obtained on a built-in type, a *version* or copy of the object is made, and the operations of this action are directed to this copy, as opposed to the global data. This

new version of the object is retained if the action goes on to commit, otherwise it is forgotten (this is analogous to the use of intentions lists described in section 1.2.2).

As previously mentioned in the Introduction, in some cases users may find that serializability is too severe a concurrency constraint to inflict upon actions that access certain shared abstract types. Consequently, Argus also provides support for extended transactions on nonserializable, user-defined abstract types such as a *semiqueue.* If a user had no choice but to rely on the *built-in atomic array* data type provided by the Argus system to model a *semiqueue* , the potential increase in concurrency could not be exploited. The *Enqueue* and *Dequeue* invocations would consist of a WRITE and a READ-WRITE combination of operations, respectively, and consequently these operations would be mutually exclusive, even if they were performed on seperate items. That is, a built-in atomic array type does not provide support for individually lockable elements.

It is assumed that new, user-defined atomic types like *semiqueues* can be defined in terms of Argus' built-in atomic types. Thus, the actual implementation of a user-defined atomic type consists of a combination of *atomic* and *nonatomic* objects, where nonatomic objects contain information that is unrestrictively available to concurrently executing transactions, and atomic objects are used to hold information neccessary for the correct interpretation of this nonatomic data. To support this use of built-in atomic objects, Argus provides specialized extensions to their regular operations. These extensions enable operations to determine whether

an action which modifies a nonatomic component of a user-defined atomic object has*committed, aborted,* or is still *active.* Depending on which of these three states an action is in, its modified version can be *available, ignored* or *withheld* from other actions, respectively.

In order to exploit the type-specific properties of abstract objects and allow for the increase in concurrency that knowledge of these semantics can permit, Argus exploits the distinction between *action* concurrency and *process* concurrency. *Action* concurrency refers to actions (which consist of one or more operations) that are considered to be *active* simultaneously, whereas *process* concurrency is used to depict two or more processes (which represent executing operations) executing on the same object at the same time. "Coarse-grained" *action* concurrency is more crucial than "fine-grained" *process* concurrency with respect to performance issues. This can be attributed to the fact that actions have the potential to take a very long time -- thus there are more serious consequences if they are permitted to exclude each other than there are if (typically smaller) processes lock each other out. Locks on built-in atomic objects are used to synchronize actions, whereas process synchronization is accomplished by means of a *mutex* data type. A mutex object is merely a container for another object and it enforces mutual exclusion of the operations that access the object it contains. For example:

**mutex**[array[int]]

is a mutex object which contains an array of integers. Consequently, all operations

performed on this array will necessarily behave exclusively.

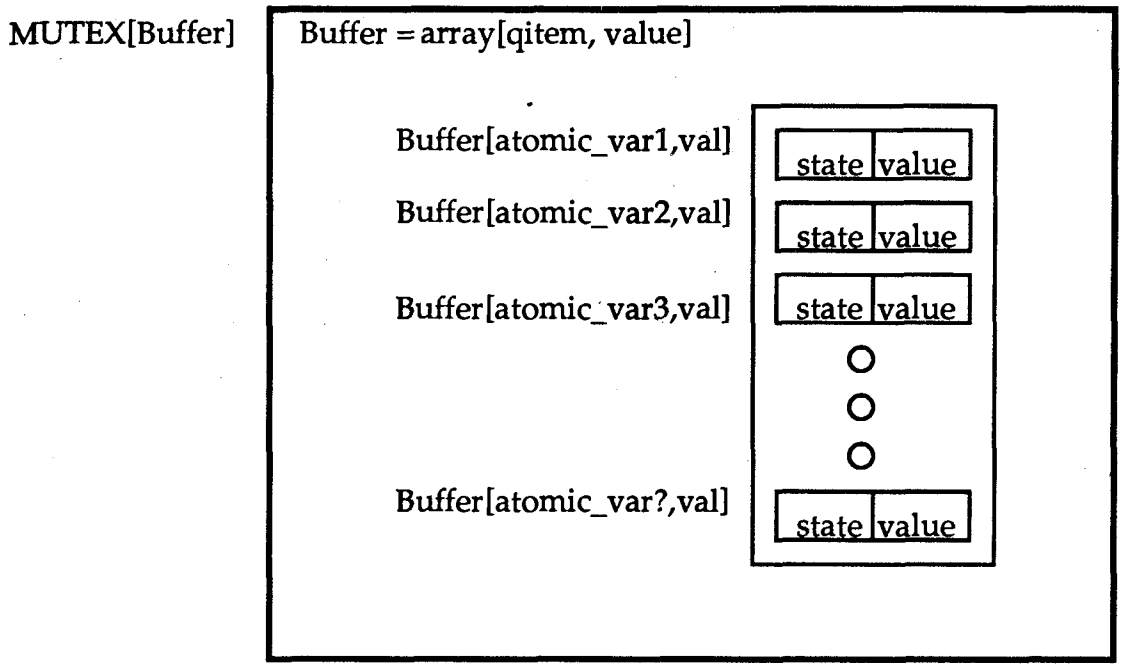
A process can take control of a mutex object using the *seize* statement. If some other process has possession of the object, this process waits until that possession is released. Although this wait should be relatively short, it introduces a further level where deadlock could potentially arise (in the event that each concurrent process attempts to seize more than one mutex object). Consequently, a run time check for this type of deadlock must be performed every time a process is forced to wait for possession of a mutex object. If several processes are waiting, one is selected fairly. Once a process has *seized* a mutex object, it is possible to release possession temporarily with a *pause* statement. After a process executes a *pause*, it waits for a system determined amount of time and then attempts to regain possession. The following high-level description of a *semiqueue* implementation will help to demonstrate the role built-in atomic types and mutex objects can play in the representation of user-defined atomic types.

Within a user-defined atomic type, built-in atomic types are used to determine the state of an action (i.e., commited, aborted, or active) and mutex objects are used to synchronize user processes (i.e., operations). In the case of a *semiqueue* of integers in Argus, one effective implementation strategy is to enqueue the integers in a nonatomic array enclosed by a mutex object [Weihl85] . This way, concurrent actions accessing the array, which have the potential to be long, need not always shut each other out -- however, the processes that execute the operations on behalf of

these actions, which will be relatively short, are mutually exclusive. Hence, any operation that reads or modifies the array must first *seize* control of it.

The built-in atomic type, *atomic_variant*, is associated with each integer in the mutex array to record the status of the action that inserted or deleted that element. Atomic_variants can be in one of two states, "present" or "absent", which are identified by a "tag" that has an associated value. If an element's state is "present", then the associated value is the enqueued integer. The value of an element in an "absent" state is ignored. Since concurrency control for the atomic_variant is governed by Argus' two phase locking technique, any modifications made by an action to the state of an element in the semiqueue are strictly local until that action commits. Conflict only occurs when an action attempts to *Dequeue* an element which has been either *Enqueued* by a currently active action or accessed by a concurrent *Dequeue* operation. Technically, this *semiqueue* can be represented as (Figure 2.1):

```
mutex[buffer]
 where  buffer == array[qitem,value]
          qitem == atomic_variant[present: int, absent: nil]
          value == integer
```




The semiqueue is modeled by an array, "Buffer", of atomic_variants, each containing <state> and <value> data. This array is contained within a mutex object, hence operations to access the array are mutually exclusive.

**Figure 2.1**: A Semiqueue in Argus

*Enqueue* and *Dequeue* operations execute in the following manner. An *Enqueue* operation first associates a new atomic_variant object with the incoming value, then *seize* s the mutex object and adds this new item to the array (it may have to wait for another operation to release the mutex object). The tag associated with this value becomes "present" if the action performing that *Enqueue* operation later commits, and is permanently set to "absent" if it aborts. A *Dequeue* operation *seize* s the mutex object and then searches for an eligable element, one of which is

selected arbitrarily. This element's tag is then changed to "absent", and its associated value is returned. In the event that a *Dequeue* operation is unsuccessful in finding an available element, a *pause* statement is executed and the search is resumed at a later time. If the action calling the *Dequeue* operation commits, then the tag is set to "absent" permanently, otherwise, if the calling action aborts, the *qitem* 's state is returned to "present". *Qitem* s with a state which has been permanently set to "absent" by either an aborted *Enqueue* or a committed *Dequeue,* are removed from the array by a garbage collection routine.

It is important to notice that the array-type in the mutex object is not atomic. Consequently if any changes were inflicted upon the integers of the array by actions that later aborted, these modifications could not be undone. Thus if a *Dequeue* operation were to actually over-write the integer value of an item in the array (instead of just changing its status), this operation could not be reversed if the calling action later aborted. *Qitems* on the other hand, which are the built-in atomic types associated with each integer of the array, are implemented such that any changes made by an action to the status of an element can be undone in the event that the action does not commit (i.e., local modifications can be thrown out).

### 2.2.2 TABS

The creators of the TABS prototype cite the following two reasons for choosing a pessimistic locking scheme to accomplish concurrency control [Spector85]:

(1) Locking has been proven to be an efficient synchronization mechanism in many commercial database management systems.

(2) When locking is local to servers, it can be customized to provide better performance. For example, *type-specific* locking [Schwarz84] .

Type-specific locking allows for more concurrency than traditional READ/WRITE locking by using semantic information associated with abstract types to define type-specific locking modes and customized compatibility relations. Unfortuately, however, TABS is currently restricted to traditional READ/WRITE modes.

In TABS, a semiqueue can be implemented as an array of *individually lockable* elements, bounded by *HEAD* and *TAIL* pointers (Figure 2.2). Since items may be removed from anywhere within this queue, gaps between these pointers must be identifiable. In order to accomplish this, all array elements have an associated boolean value, *InUse* , which indicates whether that element actually contains a value that is currently being stored in the semiqueue. *Enqueue* and *Dequeue* set and clear this bit, respectively. *Enqueue* adds a new element to the semiqueue by writing the integer below the *TAIL* pointer, setting the *InUse* bit to true, and reassigning the *TAIL* pointer to this new element. *Dequeue* scans the array for the

first eligibile element, returns its value and removes it by clearing the *Inuse* bit. Elements that are not considered eligible are either locked or have *InUse* set to false. A garbage collection routine eventually removes these elements from the array.

These *Enqueue* and *Dequeue* operations are implemented as coroutines in the semiqueue data server. Each incoming request is treated as a seperate coroutine invocation, and a coroutine switch is made when an operation is forced to wait for a lock. The semantics of these coroutines ensure that only one transaction at a time can update the *TAIL* pointer, and provide a service that is similar to that of the *mutex* object in Argus.

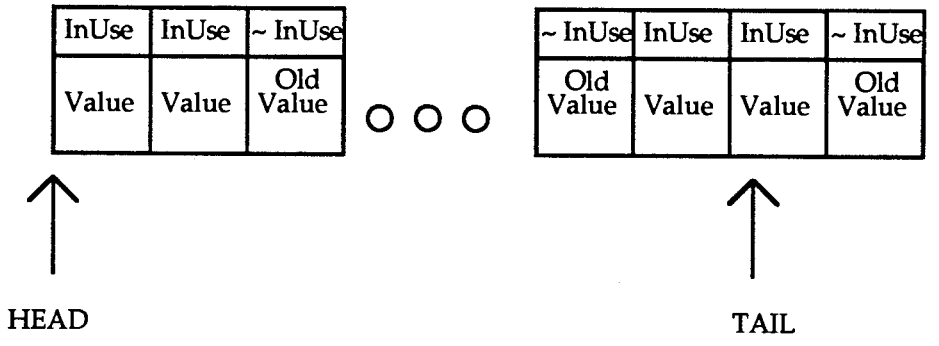



**Figure 2.2**: A Semiqueue in TABS

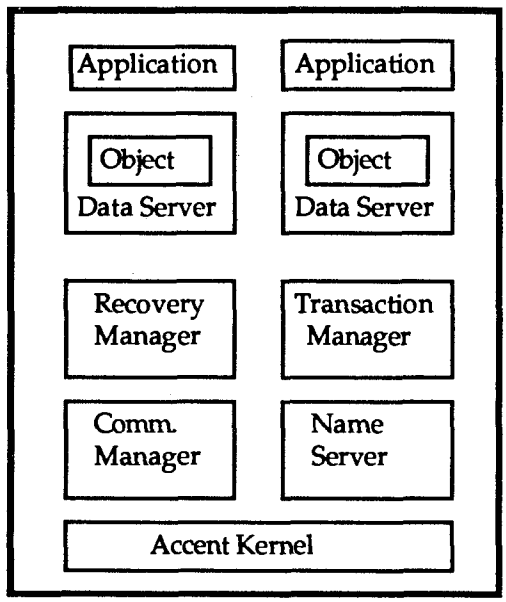
The amount of concurrency that this implementation allows with respect to *Enqueue* and *Dequeue* operations is identical to that of the Argus implementation, and can be demonstrated by the following conflict table (where *id* and *id'* represent distinct items):

| | Enqueue(*id* ) | Dequeue(*id* ) |
|---|---|---|
| Enqueue(*id* ) | <na> | X |
| Enqueue(*id* ') | | |
| Dequeue(*id* ) | X | X |
| Dequeue(*id* ') | | |

As this table suggests, the fact that TABS allows a semiqueue to be implemented as individually lockable elements permits the uninhibited concurrent execution of all transactions that access different items in the semiqueue. In the event that a transaction aborts, its effects are undone by backtracking through a log kept by the system and restoring the appropriate semiqueue elements to their previous state.

### 2.2.2.1 Components of TABS

Within the experimental design of a TABS, four basic components provide support for distributed extended transactions: recovery management to handle hardware failures, transaction management for implementing commit protocols and allocating globally unique transaction identifiers, communication managment to handle access to the network, and name dissemination for locating objects in the system. A distributed transaction can then access objects at more than one site and still be an atomic unit of work . The system components of a TABS node can be illustrated as follows (Figure 2.3):




TABS is implemented in Pascal running on a modified version of the Accent operating system. "Applications" initiate transactions and call data servers to perform operations on objects. Data servers are programmed with aid of system libraries for doing synchronization, recovery and commit.

TABS system components include: Recovery Manager for recovery and log management; Transaction Manager for handling transactions; Communication Manager for network communication and a Name Server for name dissemination.

**Figure 2.3**: The Basic Components of a TABS Node

Within the context of a system such as TABS, our work focuses on the synchronization issues that are local to a single data server.

## 2.3 Serial Dependency Relations: A Tool for Defining Type-Specific Concurrency Control

*Serial dependency relations* [Spector83, Schwarz84, Herlihy86] provide a means of incorporating semantic information into the specification of consistency constraints for type-specific operations. A *serial dependency* defines the order in which two concurrently executing transactions operate on a common object. In a general purpose system, a set of *proscribed serial dependencies* can be defined for each shared object type and used to identify type-specific conflicts. This section is dedicated to an informal examination of the notation and use of dependency relations to define type-specific consistency constraints. Formal and extensive treatment for optimistic and pessimistic methods can be found in [Herlihy86] and [Schwarz84], respectively.

Let the equation: $D: T_i:X \rightarrow_o T_j:Y$ represent the dependency D, that is formed when transaction $T_i$ performs operation $X$, and transaction $T_j$ subsequently performs $Y$ on the same object $o$. The set of ordered pairs of transactions $\{(T_i,T_j)\}$ for which $D: T_i:X \rightarrow_o T_j:Y$ holds, forms a *relation*, denoted "$<_D$". If $T_i <_D T_j$, (that is, if $(T_i, T_j)$ are one of the ordered pairs in the set that denotes the relation "$<_D$") then this means that $T_i$ *precedes* $T_j$ and $T_j$ *depends on* $T_i$ under the dependency D.

A set of proscribed serial dependencies can be used to define conflicts between

operations that use semantic information in the following way. Consider the limited semantics of the standard definitions of READ and WRITE operations within a conventional database. In total, there are four possible dependencies between a pair of transactions that access a common object [Schwarz84]:

| | |
|---|---|
| **D1:** $T_i$: READ $\rightarrow_o$ $T_j$: READ | $T_i$ reads an object subsequently read by $T_j$ |
| **D2:** $T_i$: READ $\rightarrow_o$ $T_j$: WRITE | $T_i$ reads an object subsequently modified by $T_j$ |
| **D3:** $T_i$: WRITE $\rightarrow_o$ $T_j$: READ | $T_i$ modifies an object subsequently read by $T_j$ |
| **D4:** $T_i$: WRITE $\rightarrow_o$ $T_j$: WRITE | $T_i$ modifies an object subsequently modified by $T_j$ |

Of these four, dependencies of type **D1** are considered to be *insignificant* due to the fact that they do not affect the outcome of transactions, and hence their ordering does not affect serializibility. That is, given a pair of transactions $T_i$ and $T_j$ that both read a shared object *o*, the semantics of the READ operation are such that no transaction can determine whether $T_i <_{D1} T_j$ or $T_j <_{D1} T_i$. Hence, the set of *proscribed* dependency relations for these operations is {**D2, D3, D4**} (which correspond to the definition of conflicts among READ/WRITE operations presented in section 1.2.1).

When proscribed dependencies are used to define conflicts, serializibility can be redefined as *orderability* (i.e., the property of being cycle free) with respect to the union of the set of proscribed dependency relations. For example, a READ/WRITE concurrency control scheme allows for multiple readers by permitting cycles to form

in the $<_{D1}$ relation, while preventing cycles in the union of $<_{D2}$, $<_{D3}$ and $<_{D4}$.

Consider the following two serializible schedules of operations [Schwarz84]:

| | |
|---|---|
| T1: READ(object_1) | T2: READ(object_1) |
| T2: READ(object_1) | T1: READ(object_1) |
| T1: WRITE(object_1) | T1: WRITE(object_1) |

In the first schedule, T1 $<_{D1}$ T2, and T2 $<_{D2}$ T1, which forms a cycle in the relation $<_{D1 \cup D2}$, but the relation $<_{D2 \cup D3 \cup D4}$ cycle free. The second schedule reverses the first two steps, thus removing the cycle, and still has an identical effect on the system's state. Consequently, both schedules are orderable with respect to $<_{D2 \cup D3 \cup D4}$.

In contrast, consider the two schedules:

| | |
|---|---|
| T1: READ(object_1) | T2: WRITE(object_1) |
| T2: WRITE(object_1) | T1: READ(object_1) |
| T1: WRITE(object_1) | T1: WRITE(object_1) |

The first can be shown to violate serializibility due to the fact that it is not orderable with respect to $<_{D2 \cup D3 \cup D4}$. Here, T1 $<_{D2}$ T2 and T2 $<_{D4}$ T1, thus it contains a cycle in $<_{D2 \cup D4}$, which is a subrelation of $<_{D2 \cup D3 \cup D4}$. The second schedule reverses the first two steps, removing the cycle, and the resulting schedule is orderable with respect to the union of the set of proscribed dependencies.

Serial dependency relations are used to define type-specific conflicts in each of our optimistic, pessimistic and hybrid implementations. As demonstrated in Section 3, each of these three approaches to concurrency control requires different dependencies to be included among their proscribed set of relations.

## 2.4 Optimistic Techniques: Conflict-based Validation

*Confict-based validation* [Herlihy86] is the technique that forms the basis of our optimistic implementation. In keeping with traditional approaches to optimistic concurrency control, a *conflict-based* approach works by defining each abstract data object as being a composite of two components: a *permanent state* and a set of *intentions lists*. An object's permanent state records the effects of transactions which have successfully terminated and there is an intentions list, recording tentative changes, for each active transaction which has accessed the object. When a transaction commits, changes in its intentions list are applied to the permanent state of the object. Before a transaction may commit, however, it must be validated.

Given that an *event* consists of an operation invocation and a response, *conflict-based validation* is based on predefined conflicts between pairs of *events*. For example, consider an abstract data type, *Account*, and its two associated operations *Credit* and *Debit*, which increase and decrease the balance of *Account*

by a specified amount, respectively. If the amount specified to be debited exceeds the balance of *Account* , an *Overdraft* is signalled and the balance is left unchanged. The following sequence of events (called a *history* ), demonstrates how an object's state can be modelled:

Credit($5)/Ok()
Credit($6)/Ok()
Debit($10)/Ok()
Debit($2)/Overdrawn()

Conflicting events for this data type would be the following (where "X" indicates a conflict and events in the columns are executed prior to the corresponding events in the rows):

| | Credit/Ok | Debit/Ok | Debit/Over |
|---|---|---|---|
| Credit/Ok | | | |
| Debit/Ok | | X | |
| Debit/Over | X | | |

As this table indicates, there are only two types of conflicts among these events. A successful *Debit* (Debit/Ok) conflicts with a prior successful debit, and an *Overdraft* (Debit/Over) conflicts with a prior credit . These conflicts are defined by the proscribed serial dependency relations that exist between these events (section 2.3). In other words, the "success" of a *Debit* operation is dependent upon prior *Debit/Ok* events. Consequently, a *Debit/Ok* event executed by validating transaction can invalidate *Debit/Ok* events that were performed by other actively

executing transactions. Likewise, an *Overdraft* signal will depend on prior *Credit/Ok* events since a committed addition to the balance could, potentially, invalidate an overdraft condition.

If concurrent transactions accessing the same shared data object do not contain any conflicting events, then none of their events are invalidated and hence their intentions lists will be applied to the permanent state of the object in the order that they validate (this ordering can be imposed either by assigning unique transaction numbers or using a system of logical clocks and timestamps).

Employing *events* as the units of conflict, in contrast to the mere *invocations* which pessimistic schemes typically employ, has a direct effect on concurrency. In most pessimistic schemes, a lock must be acquired before invoking an operation, thus conflicts are defined between invocations as opposed to complete events. By basing validation on events, the additional information from an invocation's results can be used to validate more interleavings than would be legal in most pessimistic techniques, and consequently allow for an increased level of concurrency. For example, within an event based model, a *Credit* operation is compatible with a successful *Debit* operation, but not with an attempted *Debit/Overdraft*. Within an invocation based model, there would be no distinction between a successful and an unsucessful *Debit* operation. A comparision of the following two tables demonstrates the dramatic contrast between the number of conflicts that exist for the *Account* data type in an event based approach as opposed

to an invocation based model:

*Event/Event:*

| | Credit/Ok | Debit/Ok | Debit/Over |
|---|---|---|---|
| Credit/Ok | | | |
| Debit/Ok | | X | |
| Debit/Over | X | | |

*Invocation/Invocation:*

| | Credit/Ok | Debit/Ok | Debit/Over |
|---|---|---|---|
| Credit/Ok | | X | X |
| Debit/Ok | X | X | X |
| Debit/Over | X | X | X |

Consequently, an event based model can support an enhanced level of concurrency over invocation based models because it exploits information about the state of an object in much the same way as Fast Path (section 2.1).

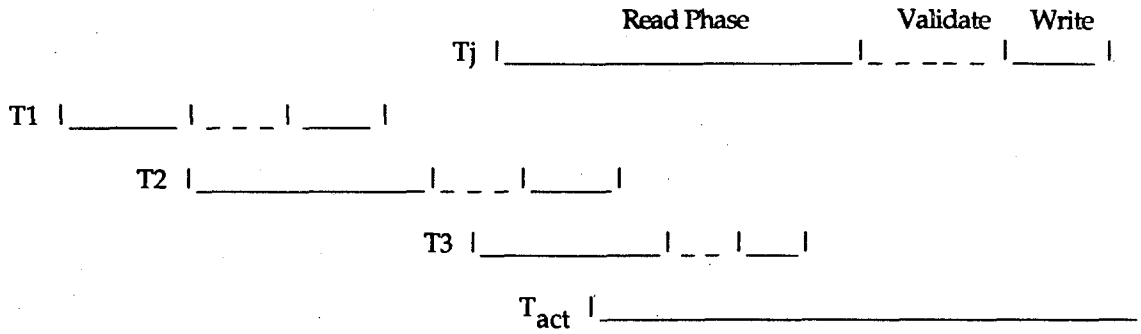
### 2.4.1 The Validation Phase

Within an optimistic scheme, concurrency control techniques are employed during a transaction's *VALIDATION* phase. Validation techniques can be classified into two categories [Harder83]: *backward* and *forward* oriented concurrency control (BOCC and FOCC, respectively). Basically, given transaction Tj that is trying to validate, BOCC checks for conflicts between Tj and transactions that

were concurrently executing with Tj that have previously validated. If there are any conflicts of this kind, then Tj has been invalidated and must be aborted. With this same Tj, FOCC would check for conflicts between Tj and transactions that were concurrently executing with Tj that are still active. If there are any conflicts of this kind, then several courses of action can be taken, the simplest of which is to abort transaction Tj. The following explanations of BOCC and FOCC rely on unique transaction identification numbers, assigned at the end of each transaction's *READ* phase, to impose a total ordering. Transactions are assumed to enter the *Validation* and possible subsequent *WRITE* phase one at a time [Harder83].

### 2.4.1.1 Backward Oriented Concurrency Control

In BOCC, given a transaction Tj that is attempting to validate, all transactions that were running concurrently with Tj, but have subsequently validated, must be checked for conflicts. That is, for the transaction Tj with transaction number *T(j)*, and for all transactions Ti with transaction numbers *T(i)*, such that $T(i) < T(j)$, (i.e., each Ti finished its *READ* phase before Tj) backward validation ensures that Tj did not access any object that was later overwritten by any Ti. The following diagram illustrates Tj's validation scenario:

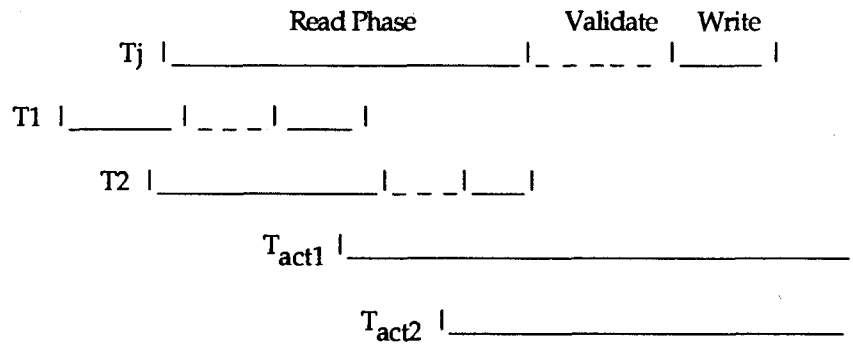



In the situation depicted above, applying BOCC during Tj's validation phase means that the objects accessed during the read phase of Tj, known as the *read set* of Tj, have to be checked against the objects written by T2 and T3, known as the *write set* of T2 and T3. If Tj's *read set* has any objects in common with either of these *write sets*, then the only solution is to abort Tj (since T2 and T3 have already been validated). This implies that in order to facilitate BOCC, all the *write sets* of a group of concurrently executing transactions must be kept until the last member of that group has committed.

### 2.4.1.2 Forward Oriented Concurrency Control

If the scheduler is attempting to validate Tj using FOCC, instead of comparing Tj's *read set* with the *write sets* of validated transactions as in BOCC, active transactions are checked for conflicts. In FOCC, the *write set* of a transaction being

validated is compared with the *read sets* of all concurrently executing transactions that not only started their execution subsequent to the beginning of Tj's execution, but also have not yet completed their *READ* phase. Consequently, concurrency control is ultimately tied to transactions that perform WRITE operations, and READ-only transactions can be validated automatically. For a transaction Tj that is trying to commit, the following scenario captures a FOCC scheme:




In the above diagram, the *write set* of the transaction seeking validation, Tj, must be checked against the *read sets* of active transactions that began execution subsequent to the start of Tj, which are $T_{act1}$ and $T_{act2}$. In the event that Tj's *write set* has an object in common with one or both of these *read sets*, one of several conflict resolution strategies could be applied [Harder83]. The most direct of these is an "immediate abort" approach, where the transaction that is trying to be validated, Tj, is aborted when such a conflict is detected.

It is important to note that since FOCC must examine potentially dynamic *read sets* of concurrently active transactions, validation can be intricate. A

straightforward approach, however, is to perform validation in a system-wide critical section [Harder83].

## 2.5 Hybrid Concurrency Control for Extended Transactions

Type-specific two phase locking (section 2.2.1) and conflict-based validation (section 2.4) are the respective pessimistic and optimistic components of a recently introduced hybrid concurrency control method for extended transactions which forms the basis of our hybrid implementation [Herlihy86]. Within conflict-based validation, there appears to be no substantial difference with regards to performance issues between FOCC and BOCC [Herlihy86]. We chose FOCC because its validation is inherently more optimistic due to the fact that its "immediate abort" strategy could abort a transaction due to a conflict with a transaction which, in turn, is not guaranteed to be validated (i.e., there could be more cyclic starts). Within our simulation, this approach's extreme optimism serves to be an appropriate counter part to the pessimistic approach for comparison purposes.

A hybrid synchronization mechanism supports the selective application of optimistic or pessimistic control on a per operation basis for an abstract object in a distributed system. Within this approach, the appropriate concurrency control mechanism can be applied according to a probability of occurance that can be

expected for each type of conflict. Conflicts that have a high probability of occuring can be governed by a pessimistic technique, whereas conflicts that are expected to occur at a sufficiently low frequency could rely on an optimistic method. The following is a high level discription of the methodology by which this type of mixed control could be accomplished.

Serial dependency relations are used to define conflicts between pairs of events. The optimistic component of this compound approach, forward oriented conflict-based validation [Herlihy86], relies on these conflicts to determine which, if any, transactions should be aborted. In the same way, the pessimistic component of this approach, type-specific two phase locking [Schwarz84, Herlihy86], can employ these conflicts to determine which transactions should be delayed. For instance, as previously established (section 2.4), the serial dependency relations for the events associated with the *Account* data type define conflicts between the events *Debit/Ok* and *Debit/Ok*, as well as between *Credit/Ok* and *Debit/Overdraft*. Within a hybrid approach, it is possible to make further distinctions about how each of these types of conflicts is to be controlled. That is, given an *Account* data object where the balance is expected to cover all debits, but for which concurrent debits are frequent, the probability of conflicts between two *Debit/Ok* events can be expected to be high, while conflicts between *Credit/Ok* and *Debit/Overdraft* events will occur with a much lower frequency. In this case, a cost-effective hybrid scheme could treat the first type of conflict pessimistically while the latter could be treated optimisitically.

This situation can be depicted as follows (where "P"= pessimistic conflict and "O"= optimistic conflict replace the generic "X" conflict that was used in section 2.4):

| | Credit/Ok | Debit/Ok | Debit/Over |
|---|---|---|---|
| Credit/Ok | | | |
| Debit/Ok | | P | |
| Debit/Over | O | | |

When a transaction executes an event that is governed by an optimistic conflict, it is given an *optimistic lock (O-lock)* for that event. A transaction can only be validated if no other transaction holds an optimistic lock for a conflicting event. That is, given a transaction $T_i$ that is attempting to validate (via FOCC) and holds an *O-lock* for the event *Credit/Ok*, and an active transaction, $T_j$, which holds an *O-lock* for the event *Debit/Overdraft*, $T_i$ must be aborted. All O-locks are released upon transaction termination.

In contrast, *P-locks* must be requested after a transaction executes an operation invocation, but before it updates its intentions list. If any other transaction holds a conflicting *P-lock*, the lock is refused and the operation must be retried. That is, given a transaction, $T_i$, which is attempting to execute the event *Debit/Ok*, and a transaction, $T_j$, which already holds a *P-lock* for the event *Debit/Ok*, $T_i$ must be delayed and its operation invocation retried. When a P-lock can be successfully obtained, the intentions list is updated and the event's response

returned to the calling transaction. As with O-locks, all P-locks are released upon transaction termination.

## 2.6 Performance Evaluation

Numerous studies have been dedicated to examining the performance of concurrency control algorithms for conventional database systems [Thanos83, Franaszek85, Agrawal87, Pun87, Wolfson87]. A common complaint associated with studies like these is that their results often appear to be contradictory and inconclusive. As suggested in [Agrawal87], the discrepancies that arise between different evaluations can be attributed to the fact that every study is based on its own unique set of performance modelling assumptions. In order to allow for a comparison between our results and other studies, we first had to establish a common and acceptable performance metric.

Presently, studies on the performance of concurrency control mechanisms that exploit semantic knowledge are sparse. One such study [Cordon85] compares the performance of a concurrency control scheme that utilizes application specfic semantics to conventional two phase locking. The purpose of this comparison is to determine the conditions under which the higher complexity of overhead associated with the application dependent mechanism pays off. This work brings attention to

the possibility that the cost of a customized mechanism may outweigh its potential gains in concurrency if its overhead is much higher than that of conventional methods.

Through the use of a simulated distributed database management system model, this work has established that the *level of conflict* was the most dominant performance factor in the comparison. In an environment such as a large database where references are widely dispersed and probability of conflict is consequently low, conventional two phase locking schemes outperform an application dependent strategy. This is due to the fact that the conventional approach has smaller overhead. But for environments in which the probability of conflict is expected to be high, application dependent methods can significantly improve the system's response time. The reason for this is the increased amount of concurrency schemes such as these can support.

Similarly, as previously discussed (section 1.2.3), the *level of conflict* can be used to determine which of the optimistic or pessimistic techniques is most cost-effective. As concluded in [Kung81], optimistic methods are superior to traditional locking methods in environments where transaction conflict is rare, but inferior where transaction conflict is more frequent.

Relying on the level of conflict as a suitable performance metric and amalgamating the conclusions of [Cordon85] with [Kung81] results in the following expectations for the ordering of conventional locking (PESS), locking based on

semantic knowledge (SK), and optimistisic (OPT) methods of concurrency control:

| | *Performance* | | |
|---|---|---|---|
| | Best | Mediocre | Worst |
| *Level of Conflict* | | | |
| **High:** | SK | PESS | OPT |
| **Low:** | OPT | PESS | SK |

**Figure 2.4**: Expected Performance under High and Low Levels of Conflict

In keeping with these studies, we use the percentage of conflict as our performance variable within our transaction simulation model. Subsequently, we examine the relative behaviors of the optimistic, pessimistic and hybrid semiqueue servers as they are subjected to an increasing percentage of conflict. Since this, to our knowledge, is the first implementation of the techniques presented in [Herlihy86], we are specifically interested in establishing some notion of a *threshold* percentage of conflict for these methods. In accordance to the results previously construed from [Cordon85] and [Kung81] (Figure 2.4), there should be some level of conflict where the performance of locking based on semantic knowledge and optimistic approaches intersect. Below this threshold percentage, the optimistic technique outperforms semantic based (type-specific) locking, whereas above the threshold, type specific locking becomes superior. By close examination of this and other implementation dependent performance results, we establish some characteristics that generalize to applications of these techniques to other abstract data types.

# 3 An Example: The Semiqueue Server

A *semiqueue* is one example of a shared abstract object that has been employed by existing extended transaction facilities (Argus and TABS, section 2.2) to demonstrate their ability to support an increased level of concurrency over traditional facilities. As previously established, a queue of this kind allows for more concurrency than strictly FIFO queues because the ordering of its elements is not important. The events we defined for our semiqueue include: *Enq(item)/Ok()*, which simply enqueues an "item" and returns an "ok" response; *Deq()/Ok(item)*, which returns an "item" from the semiqueue; *Deq()/Failed()* , which indicates that the semiqueue is empty; and *Eval()/Ok(#items)*, which counts the number of items in the semiqueue and returns the total . An outline of our optimistic, pessimistic and hybrid semiqueue servers and their respective use of dependency relations to identify conflicts is presented in the sections that follow.

## 3.1 The Optimistic Server: Four Proscribed Dependency Relations

In the context of Herlihy's optimistic event-based synchronization scheme, the following serial dependency relations define conflicts for semiqueues (where the events in the columns are executed prior to the events in the corresponding rows):

| | Enq(i)/Ok | Enq(i')/Ok | Deq/Ok(i) | Deq/Ok(i') | Deq/Failed | Eval/(#items) |
|---|---|---|---|---|---|---|
| Enq(i)/Ok | na | | na | | | |
| Deq/Ok(i) | na | | X3 | | na | |
| Deq/Failed | X1 | X1 | na | na | | |
| Eval/(#items) | X2 | X2 | X4 | X4 | | |

As this table indicates, there are four proscribed dependency relations, reflecting the facts that an *Enq(i)/Ok()* event will invalidate both a *Deq()/Failed()* and an *Eval()/Ok(#items)* event (X1, X2), and a *Deq()/Ok(i)* event will invalidate both a *Deq()/Ok(i)* (where *i* represents one unique element) and an *Eval()/Ok(#items)* event (X3, X4). More formally, the proscribed set of dependencies can be represented as:

**D1**: $T_i$:Enq(i)/Ok() ->$_q$ $T_j$:Deq()/Failed() (X1)
**D2**: $T_i$:Enq(i)/Ok() ->$_q$ $T_j$:Eval()/Ok(#items) (X2)
**D3**: $T_i$:Deq()/Ok(i) ->$_q$ $T_j$:Deq()/Ok(i) (X3)
**D4**: $T_i$:Deq()/Ok(i) ->$_q$ $T_j$:Eval()/Ok(#items) (X4)

Given a transaction, *Ti*, that is attempting to validate (according to *forward* oriented, conflict-based validation), if there are any conflicts between the events executed in *Ti* and other active transactions, *Ti* will be aborted. For example, if *Ti* tries to execute an *Enq(i)/Ok()* event, and another active transaction has executed *Deq()/Failed()* or *Eval()/Ok(#items)*, then *Ti* is aborted. Likewise, if *Ti* tries to execute a *Deq()/Ok(i)* event, and another active transaction has executed *Deq()/Ok(i)* or *Eval()/Ok(#items)*, then *Ti* cannot be successfully validated (ie is

aborted).

The optimistic server treats all four of the conflict relations mentioned above optimisticially through the use of optimistic locks (*O-locks* ) and *intentions lists.* O-locks are automatically granted each time a transaction executes an event which could potentially be invalidated (ie those events on the right side of the dependency equations D1-D4, eg: *Eval()/Ok(#items)* ). An intentions list serves as a local buffer to record the tentative changes made to the shared object by an active transaction. A transaction, *Ti* , can be validated iff there are no transactions that hold an O-lock for an event that conflicts with an event in *Ti* 's intentions list. If *Ti* can be successfully validated, the tentative changes recorded in its intentions list are applied to the shared semiqueue and its O-locks are released.

### 3.1.1 Implementation Outline of the Optimistic Server

O-locks for *Deq()/Failed, Eval/Ok(#items),* and *Deq()/Ok(i)* events are implemented simply as boolean flags associated with the transaction number (*Tnum* ) of the calling transaction (in the case of *Deq()/Ok(i)* , there is also a flag associated with the element *i* ). Whenever a transaction executes one of these events, it is automatically granted the appropriate O-lock and then updates its intentions list accordingly. A transaction's intentions list takes the form of two

seperate lists which record the tentative changes made by *enqueue* and *dequeue* operations respectively. The first is a doubly linked list of locally enqueued items that becomes attached to the semiqueue if the transaction is successfully validated, and the second is a linked list of pointers to elements in the shared semiqueue that the active transaction "intends" to dequeue. A more in-depth look at this server's implementation details is presented in Section 4.

## 3.2 The Pessimistic Server: Seven Proscribed Dependency Relations

Once a transaction has executed an event (ie *Enq(i)/Ok, Deq()/Ok(i), Deq()/Failed()* or *Eval/Ok(#items)* ) at the pessimistic semiqueue server, it requests a pessimistic lock (P-lock) for that event before updating its intentions list. If any other transaction holds a conflicting lock, the lock is refused, the event is discarded, and the transaction must retry the invocation (which may return a different response). When a P-lock can be granted, the transaction's intentions list is updated and the response is returned. Once a transaction has successfully obtained all of its P-locks and has completed execution, its intentions list is applied to the global object and its P-locks are subsequently released.

Pessimistic conflicts are defined by a set of proscribed dependencies which, unlike the optimistic conflict relations, must be symmetric since the commit order

of transactions is unknown at the time conflicts are detected. The following table depicts the seven pessimistic conflicts that exist between semiqueue events:

| | Enq(i)/Ok | Enq(i')/Ok | Deq/Ok(i) | Deq/Ok(i') | Deq/Failed | Eval/(#items) |
|---|---|---|---|---|---|---|
| Enq(i)/Ok | na | | na | | X5 | X6 |
| Deq/Ok(i) | na | | X3 | | na | X7 |
| Deq/Failed | X1 | X1 | na | | | |
| Eval/(#items) | X2 | X2 | X4 | X4 | | |

The above conflicts consist of the original four proscribed dependency relations previously defined for an optimistic server (D1, D2, D3, and D4), along with their symmetric counter-parts (D5, D6 and D7). Hence, the proscribed set of dependencies can be denoted as follows:

**D1**: $T_i$:Enq(i)/Ok() ->$_q$ $T_j$:Deq()/Failed() (X1)
**D2**: $T_i$:Enq(i)/Ok() ->$_q$ $T_j$:Eval()/Ok(#items) (X2)
**D3**: $T_i$:Deq()/Ok(i) ->$_q$ $T_j$:Deq()/Ok(i) (X3)
**D4**: $T_i$:Deq()/Ok(i) ->$_q$ $T_j$:Eval()/Ok(#items) (X4)
**D5**: $T_i$:Deq()/Failed() ->$_q$ $T_j$:Enq()/Ok() (X5) (inversion of D1)
**D6**: $T_i$:Eval()/Ok(#items) ->$_q$ $T_j$: Enq(i)/Ok() (X6) (inversion of D2)
**D7**: $T_i$:Eval()/Ok(#items) ->$_q$ $T_j$: Deq(i)/Ok() (X7) (inversion of D4)

It is important to note that within the context of a pessimistic server, one of these conflicts, D3, should be treated differently from all the rest. This kind of conflict would arise between two transactions attempting to dequeue the same item, and thus it would be pointless to make one of them wait for the other to finish executing if the semiqueue has yet another item that is eligible for dequeueing.

Consequently, the state of the semiqueue can be used to determine whether or not conflicts of this kind necessarily introduce a delay (this is similar to the treatment of dequeue operations in both Argus and TABS).

### 3.2.1 Implementation Outline of the Pessimistic Server

As with the optimistic implementation, the pessimistic server uses a doubly linked list for the shared semiqueue, and a two-part intentions list for each active transaction: an *enqueue* list, consisting of a local, doubly linked list of elements to be enqueued, and a *dequeue* list consisting of a pointers to shared items the transaction "intends" to dequeue. These lists are treated in a manner identical to that of the optimistic server.

As previously established, in situations where a transaction's request for a P-lock is denied (due to the fact that another transation already holds a P-lock for a conflicting event), the event must be discarded and the transaction must retry the invocation. In order to accomodate these delayed/blocked operations, special *wake-up* calls have been included in the protocol associated with the releasing of P-locks. For example, with an *Enq* operation, the server checks to ensure that no other active transaction holds a P-lock on either a *Deq()/Failed()* or an *Eval/Ok(#items)* event (D1 and D2 above). If there is a conflict, the *Enq*

invocation will block, waiting to receive a *wake-up* message. Wake-up messages are sent out to blocked operation invocations every time a transaction that has performed an event which conflicts with the blocked operation has released an appropriate P-lock. This way, when a transaction that executed a *Deq()/Failed()* event has released its *Deq()/Failed()* P-lock, the blocked *Enq* operation will be reactivated and retried. If this P-lock happened to be the blocked *Enq's* sole impediment, the operation proceeds, otherwise it will be blocked again and must wait for the next wake-up message.

When a transaction has successfully obtained all of its neccesary P-locks and has finished executing, it inflicts the changes recorded in its *enqueue* and *dequeue* intentions list to the global state of the semiqueue. Once these changes have been accomplished, the committing transaction's P-locks can be released and wake-up calls issued to any blocked invocations that could benefit by these releases.

### 3.2.2 Deadlock

Given two transactions, T1 and T2, and the following combination of events:

T1: Deq()/Ok(i)
T2: Deq()/Ok(j)
T1: Eval --- BLOCKED, waiting for wake-up from T2
T2: Eval --- BLOCKED, waiting for wake-up from T1

it becomes evident that some kind of deadlock prevention or resolution mechanism must be included in this pessimistic server. Currently, a prevention mechanism for simple deadlock that can arise between two transactions has been implemented. This mechanism is invoked just before a transaction blocks, and given T1 that is about to block on T2, enforces the following rule:

> Deadlock Rule: For T1 to block on T2, T2 can not already be blocked on a lock type that T1 possesses.

Consequently, in the example above, T2 will not wait for T1 since T1 is already blocked and waiting for the release of T2's *Deq/Ok(j)* P-lock. In this situation, T2 would be aborted and have to be restarted.

The question now remains: is this mechanism robust enough to handle deadlock between more than two transactions? The answer to this depends on the type of circular waits that can arise between transactions. We show that all circular waits of length $n$ contain a cycle of length two, so this mechanism is sufficient.

Proof: The following diagram (Figure 3.1) depicts the conflicts that exist between the four possible semiqueue events: *Deq/Failed*, *Enq(i)/Ok*, *Eval/#items*, and *Deq/Ok(i)* (where each arc = a conflict):

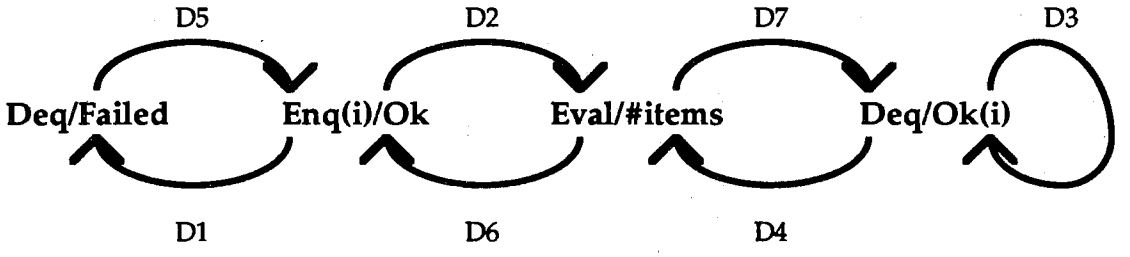


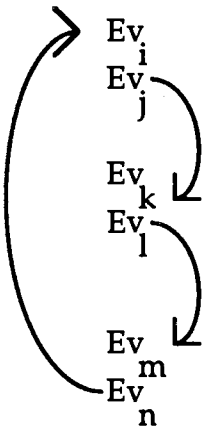

**Figure 3.1**: Pessimistic Conflict Relations

As previously mentioned, in the context of a pessimistic server these conflicts are necessarily symmetrical due to the fact that the commit order of transactions is unknown when these conficts are detected.

Consider the structure of a deadlock situation between three transactions, T1, T2 and T3, which execute the following events:

T1: $Ev_i$, $Ev_j$
T2: $Ev_k$, $Ev_l$
T3: $Ev_m$, $Ev_n$

In order for a circular wait to arise between all three of these transactions, they must all be blocked in the following manner (where $Ev_i$, $Ev_k$ and $Ev_m$ have all been completed, and $Ev_j$, $Ev_l$ and $Ev_n$ are blocked according to the arcs):




In order for a situation such as this to arise, where a circular wait developes between three transactions that does not include a cycle of length two, the following properties must hold:

- $Ev_j$ does not confict with $Ev_m$ (since this would create a cycle of length two) and therefore $Ev_m$ is not equal to $Ev_k$.
- $Ev_l$ does not confict with $Ev_i$ (since this would create a cycle of length two) and therefore $Ev_m$ is not equal to $Ev_i$.
- $Ev_n$ does not confict with $Ev_k$ (since this would create a cycle of length two) and therefore $Ev_i$ is not equal to $Ev_k$.

These properties cannot be achieved by the events associated with this particular implementation of a semiqueue since they require each blocked transaction to complete execution of one of the four possible events that has not yet been executed by the other transactions. That is, T1, T2 and T3 must successfully complete

execution of $Ev_i$, $Ev_k$ and $Ev_m$, respectively, before this circular wait can arise and since none of these events are the same, they must represent three of the four possible semiqueue events. As demonstrated in Figure 3.1, no combination of more than two distinct events can be successfully completed by concurrent transactions without causing a conflict. As a result, all circular waits of length $n$ must contain cycles of length two, and subsequently the prevention method that disallows cycles of length two from forming is adequate for our implementation.

## 3.3 A Hybrid Server: Four Proscribed Dependency Relations

The four proscribed dependency relations for an optimistic semiqueue server modelled after Herlihy's conflict-based validation technique [Herlihy86] can be illustrated as follows (Figure 3.2):

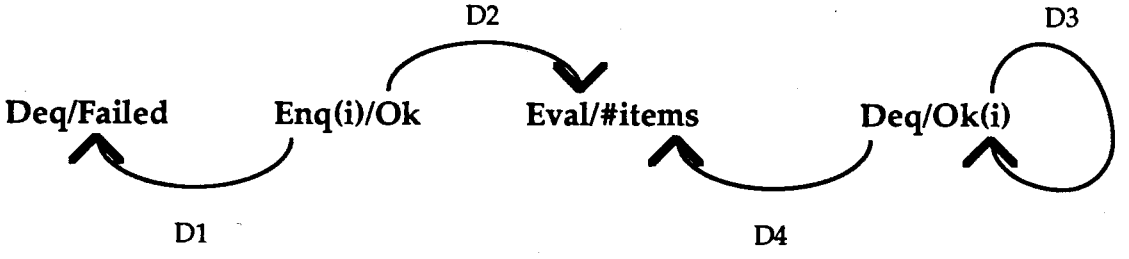



**Figure 3.2**: Optimistic Conflict Relations

A pessimistic semiqueue server, as previously discussed, must also handle the

symmetric counter parts to these conflicts, which results in the addition of relations D5, D6 and D7 (Figure 3.1).

The hybrid server considered here treats all conflicts optimistically except for the conflict defined by the dependency relation D3. This configuration was arbitrarily selected from the $2^4$ (where 4 is the number of conflicts defined by the proscribed serial dependency relations for the optimistic server) possible hybrid combinations. This particular type of hybrid server is designed to be appropriate in a situation where concurrent *Deq/Ok(i)* events are expected to be frequent, but the conflicts defined by dependencies D1, D2 and D4 are expected to be rare. Hence the hybrid conflicts can be represented as follows (Figure 3.3):

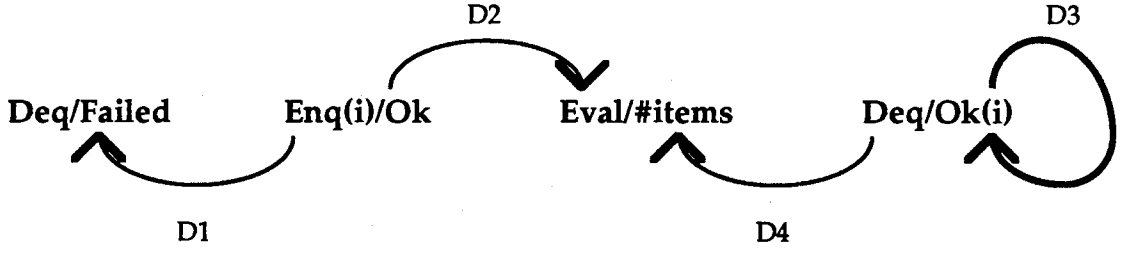



**Figure 3.3**: Hybrid Conflict Relations

In this hybrid scheme, any conflicts of type D3 are resolved pessimistically (indicated with the darker arc) by introducing delays, whereas conflicts of type D1, D2 and D4 are resolved optimistically (indicated with the lighter arcs) by transaction abort. As with the pessimistic server, the hybrid server includes a mechanism which can prevent deadlock between two transactions.

## 3.4 The Simulation Model

Our implementation simulates optimistic, pessimistic and hybrid concurrency control mechanisms for extended transactions based on the techniques presented in [Helihy86]. Although our simulation model is simple, we believe that it captures, on a small scale, the fundamental concurrency considerations addressed by both Argus and TABS. Within each of these systems a distributed program consists of a group of servers communicating via operation invocations (section 2.2). Each server completely encapsulates at least one data object and the operations that manipulate it, and consequently is responsible for controlling the concurrent access of transactions operating on its object(s). Both systems adhere to the philosophy that "fine-grained" operation concurrency is much less important than "coarse-grained" transaction concurrency and, as a result, the execution of operations within each server is mutually exclusive.

In keeping with these systems, we also employed a client/server model to support extended transactions. A full implementation of a formal transaction system however, was not appropriate at this stage. Instead, we implemented servers that could potentially operate within the type of environment defined in both Argus and TABS, and simulate the presence of a formal transaction system by "spoon-feeding" operation invocations to each server as if they were coming from active transactions. A more detailed discussion of our simulation is presented in

Section 5.

# 4 Implementation Details

## 4.1 The Semiqueue and Intentions Lists

Our implementation is done in Version 1 of the SR programming language [Andrews87, Andrews88] running under the Sun UNIX 4.2 operating system (release 3.4)[2]. Each of the optimistic, pessimistic and hybrid servers uses a doubly linked list for the shared semiqueue. Each node in this list contains an integer value (since this is a semiqueue of integers) and the appropriate optimistic or pessimistic locking information. *Qhead* and *Qtail* mark the beginning and end of the semiqueue, respectively (Figure 4.1).

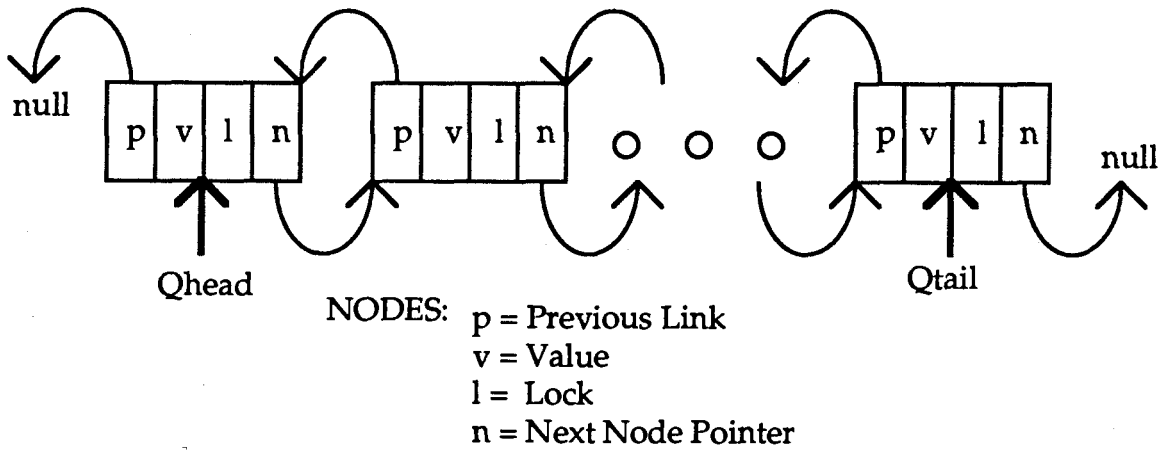



**Figure 4.1**: The Shared Semiqueue

---

[2] The source code for each of the three servers is presented in the appendix.

Within our simulation every active transaction is identifiable by a unique transaction number, *Tnum*. For every *Tnum*, each of the optimistic, pessimistic and hybrid servers maintains an intentions list. This list has two components: an *enqueue* (*Enq* ) list and a *dequeue* (*Deq* ) list.

The enqueue component consists of a doubly linked list of elements to be enqueued, and the dequeue component consists of a singly linked list of pointers to the shared elements of the semiqueue (Figure 4.2 and Figure 4.3). When a transaction successfully commits, the *Enq* list is attached to the end of the shared semiqueue, *Qtail* is updated, and the appropriate elements associated with the *Deq* component of the transaction's intentions list are removed from the semiqueue. In the event of transaction abort, however, these lists are thrown out.

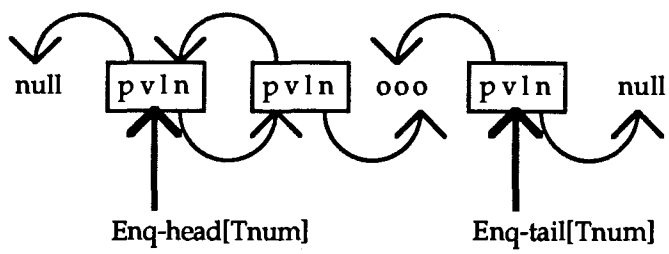



**Figure 4.2**: Enq Intentions List

The nodes of the enqueue component of each transaction's intentions list are identical to those of the shared semiqueue (Figure 4.1). This facilitates the method of appending the entire *Enq* intentions list at commit time.

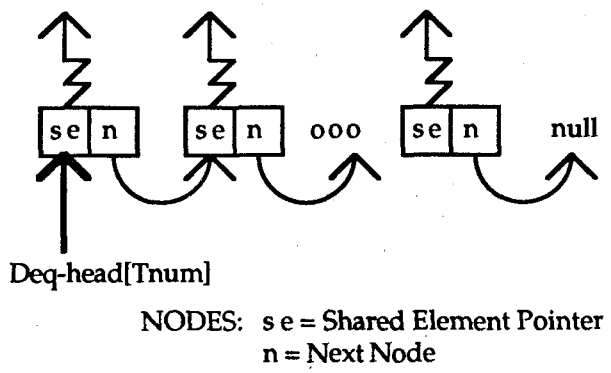



**Figure 4.3:** Deq Intentions List

## 4.2 The Operations

### 4.2.1 Optimistic Operations

Within the forward oriented optimistic server, the *Enq* (enqueue) operation is invoked with the calling transaction's *Tnum* and the integer value to be enqueued. A new node is created , assigned this value, and added to the end of Tnum's *Enq* intentions list. The response associated with this operation, which is always "OK" (i.e., the event Enq(i)/Ok), is then returned to the calling transaction. There are no optimistic locks allocated for an *Enq* event since (as discussed in section 3.1) this event can not be invalidated by any other event.

The *Deq* (dequeue) operation is also invoked with the calling transaction's *Tnum*, and either returns a value from the semiqueue (constituting the event *Deq/Ok(value)* ) or a *failed* response (*Deq/Failed* ). Given a situation where the

*Enq* intentions list associated with the calling *Tnum* is not empty, an item is simply removed from the head of this list (since *Tnum* regards these elements as being enqueued). If, however, this list is empty, the semiqueue is scanned for an *eligible* element. Any element that is not already in the *Deq* intentions list of the calling transaction is eligible. In order to easily identify which elements are in the *Deq* intentions list of a given *Tnum,* each element in the semiqueue provides Deq O-locks (associated with *Deq/Ok(value)* events) for all active transactions. These O-locks are implemented as an array[3] (for Tnums from 1 to N, where N is the maximum number of concurrently active transactions invoking the server) in the lock portion of the semiqueue nodes (Figure 4.4).

When a transaction is searching for an eligible element, it first checks its own Deq O-lock associated with that element (Deq O-lock[Tnum]). If it is not already locked, another check is made to see if any other transaction already holds an O-lock on this element (flagged by Deq O-lock[N+1] , which is set when an element is O-locked by any transaction). This additional check to ensure that the element is not already in the *Deq* intentions list of another transaction demonstrates the optimistic server's use of state based information.

---

[3] Version 1 of SR does not support boolean arrays in records, hence we were forced to use integer.

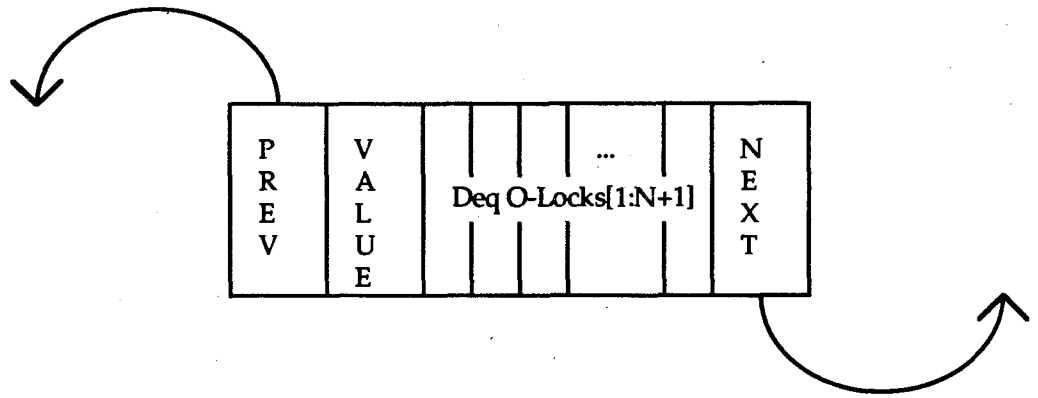



**Figure 4.4**: Deq O-locks in an Optimistic Semiqueue Node

If there is an eligible element that is not O-locked by any other transaction, then that element's Deq O-lock[Tnum] is set, the element is flagged as having been O-locked (Deq O-lock[N+1]), a pointer to the element is added to Tnum's *Deq* intentions list, and the value is returned. But, if all eligible elements are O-locked by other transactions, the first eligible element among them is selected. This selection will ultimately produce a conflict (proscribed dependency D3, section 3.1) that will be resolved during validation. If there are no elements eligible for dequeuing, a *Deq-Failed* O-lock is set for the calling transaction, and a *failed* response (constituting a *Deq/Failed* event) is returned. The server maintains one Deq-Failed O-lock for each *Tnum* (1 to N) in a boolean array (Figure 4.5).

An *Eval* operation is invoked with the *Tnum* of the calling transaction and returns the total number of elements in the semiqueue. From the point of view of the calling transaction, this total includes the elements in that transaction's *Enq* intentions list, plus the elements in the shared semiqueue, minus the number of

elements in the transaction's *Deq* intentions list. The result of this tally is returned to the calling transaction, and an *Eval/(#items)* O-lock is set for that *Tnum*. As with the Deq-Failed O-locks, the server maintains one Eval O-lock for each Tnum in a boolean array (Figure 4.5).

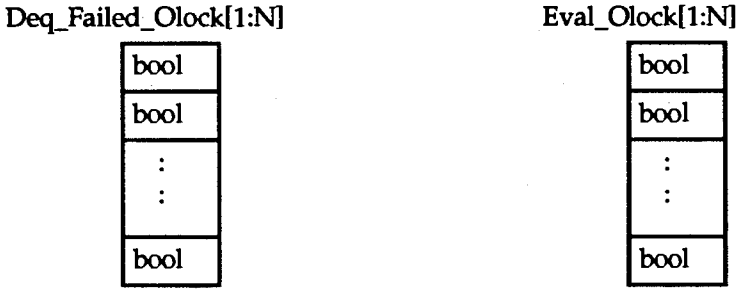



**Figure 4.5**: Deq/Failed and Eval O-locks

The*Validation* operation is also invoked with the calling transaction's *Tnum*. As with other operations, it executes with mutal exclusion and basically proceeds in the following manner:

```
Step One: (According to Proscribed Dependencies D1 and D2 section 3.1)
        if Enq Intentions list[Tnum] ≠ ø ->
                for all transaction numbers, i, ≠ Tnum ->
                    if (Deq_Failed_Olock[i] or Eval_Olock[i]) ->
                        ABORT(Tnum) and return
                    fi
                af
        fi
```

Step Two: (According to Proscribed Dependencies D3 and D4 section 3.1)

```
if Deq Intentions list[Tnum] ≠ ø ->
        for all transaction numbers, i, ≠ Tnum ->
            if ((Deq_Olock[i]  and Deq_Olock[Tnum] {for any element} or
                 Eval_Olock[i] ) ->
                ABORT(Tnum) and return
            fi
        af
fi
```

Step Three:

Commit Enq and Deq Intentions lists.
Release all O-locks held by Tnum.

Due to the fact that our simulation was contrived to test situations where the number of active transactions was approximately equal to the maximum number of concurrent transactions supported by the server, steps one and two above check all Tnums, not just explicitly active transactions (section 2.4.1.2).

The *Abort* operation called during steps one and two of validation simply discards the intentions list and O-locks associated with *Tnum*. Our simulation model does not include an automatic transaction restart mechanism.

### 4.2.1.1 Implementation Tradeoffs

Some of the implementation features of the optimistic server provide less than optimal performance. For example, instead of relying on boolean arrays to represent

transaction O-locks, an alternative approach could use linked lists of transaction identification numbers. In particular, this would greatly simplify the lock portion of each element in the shared semiqueue (Figure 4.6):

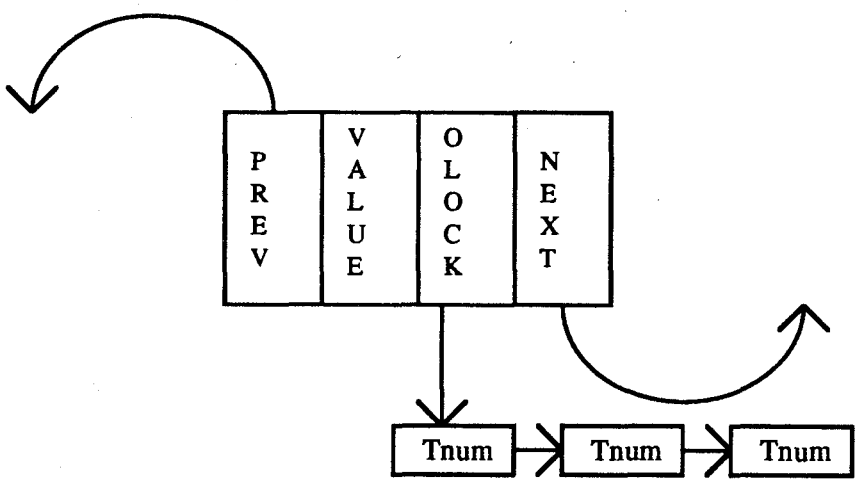



**Figure 4.6**: A Linked List of Deq O-locks in an Optimistic Semiqueue Node

The use of linked lists in this way could potentially enhance the performance of the *Enq* operation but at the same time diminish the performance of the *Deq* operation. An *Enq(i)/Ok* event could benefit by virtue of the fact that the new semiqueue node is much smaller and consequently faster to create. A *Deq/Ok(i)* event, however, could be hindered by the list-searching overhead the linked list of Deq O-locks would introduce. For example, when attempting to establish the eligibility of an element for dequeuing, the optimistic server first checks that the calling transaction does not already have a Deq O-lock set for that element. This

would require a search of the list of Deq O-locks associated with the lock portion of the element in question, and hence add to the overhead of a *Deq* operation. Low level performance tradeoffs of this kind were not, for the most part, considered to be as crucial as higher level tradeoffs.

The most important high level tradeoff we encountered deals with the optimistic integrity of this server. The fact that the states of Deq O-locks associated with transactions other than the calling transaction are taken into account when selecting an appropriate element for dequeuing does not demonstrate true optimism. Further, this "cheating" promotes the premature identification of *Deq/Ok(i) -> Deq/Ok(i)* conflict types during the READ phase of a transaction (i.e., pre-validation) in the situation where all eligible elements are already O-locked by transactions other than the calling transaction. Since selecting an element that is already O-locked by another transaction will ultimately result in transaction abort of one or the other O-lock owners, it would appear to be reasonable to abort the calling transaction during its READ phase. Although, from a performance standpoint this would potentially cut back on the amount of wasted work, from an integrity standpoint it would degenerate the optimistic server to some kind of pessimistic derivative that relies on abort as opposed to delay for conflict resolution. Consequently, even though this type of conflict can easily be identified before validation, we chose not to exploit this information in order to preserve the optimistic nature of the server and provide a more appropriate counterpart for

comparison purposes to the pessimistic server.

### 4.2.2 Pessimistic Operations

As with the optimistic server, the pessimistic server also maintains *Enq* and *Deq* intentions lists (Figures 4.2 and 4.3). When a transaction invokes an *Enq* operation, the server first checks to ensure that no other active transaction holds a P-lock on either a *Deq/Failed* or an *Eval/(#items)* event (Figure 4.7). If no such conflict exists, then it assigns the calling transaction a P-lock for an Enq(i)/Ok event, and adds the element to the local *Enq* intentions list. If, on the other hand, there is a conflict, the *Enq* invocation will block, waiting to receive a "wake-up" message (section 4.2.2.2). Wake-up messages are sent out to blocked operation invocations every time a transaction that has performed an event that conflicts with the blocked operation has released its P-locks. This way, when a transaction that executed a *Deq/Failed* event has released its *Deq/Failed* P-lock, the blocked *Enq* operation will be reactivated and retried. If this P-lock happened to be the blocked Enq's sole impediment, the operation proceeds, otherwise it will be blocked again and must wait for the next wake-up message.

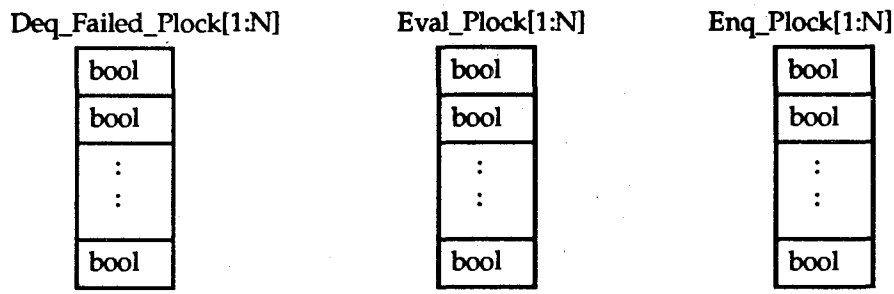



**Figure 4.7**: Deq/Failed, Eval and Enq P-locks

As with the optimistic server, a *Deq* invocation begins with a check of the calling transaction's local *Enq* intentions list. If it is not null, an item is removed from this local list (the Enq P-lock for this transaction will be released if this was the only item in the list, and invocations blocked by this type of lock will be sent a wake-up message at this time). If, however, there are no local items in the *Enq* intentions list, the transaction must access the shared semiqueue.

As opposed to the locking information associated with the nodes of the optimistic semiqueue, nodes of the pessimistic semiqueue only require one Deq P-lock (Figure 4.8). This reflects the fact that only one transaction can hold a pessimistic lock on a shared element at any given time. Since elements that have a Deq P-lock associated with the calling *Tnum* are not available for dequeuing, the first thing the server does is assess the number of eligible items. Since Deq P-locks are merely boolean values, one of the most immediate, yet admittedly crude, ways of accomplishing this was to compare the number of elements in the calling transaction's *Deq* intentions list with the number of elements in the shared semiqueue. If all eligible elements are P-locked by other transactions, *Tnum*

becomes blocked (according to the proscribed dependency D3, section 3.2) and waits for a wake-up call from either a committing transaction that has enqueued new elements, or an aborting transaction that was attempting to remove shared items.

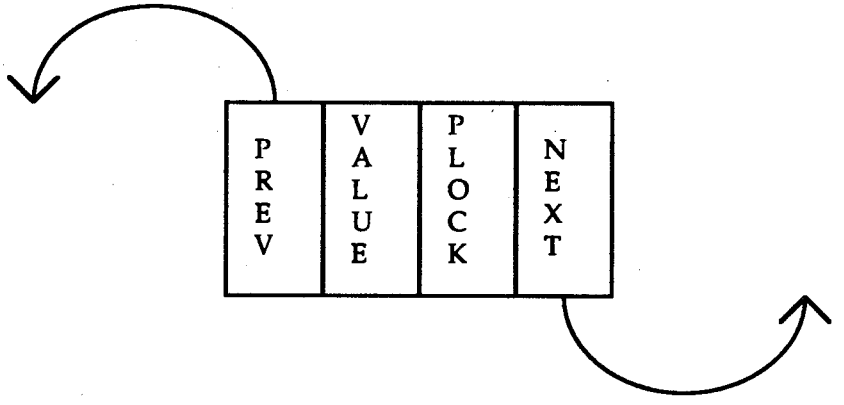



**Figure 4.8**: A Deq P-lock in a Pessimistic Semiqueue Node

If, however, there are no eligible elements in the semiqueue, the transaction needs to obtain a P-lock for a *Deq/Failed* event, and hence a check is made to ensure that no other transaction has performed an *Enq(i)/Ok* event. If no such conflict exists, the invocation is assigned a P-lock for a *Deq/Failed* event and returns a *failed* result to the caller, otherwise the operation invocation will block, and wait to receive a wake-up message.

Once this operation has been unblocked, the server re-checks the semiqueue for eligible elements. If there is one, a *Deq/Ok(i)* P-lock for this transaction must be associated with the node, and hence a further check must be made that no other transaction has performed an *Eval/(#items)* event.

Again, if there is a conflict, the invocation will be blocked, waiting to be awoken by a transaction that released an Eval P-lock. When this *Deq* is unblocked it must ensure that there still is an eligible item in the the semiqueue. If there is one, it associates a Deq P-lock with that element, adds a pointer to its *Deq* intentions list and returns the value of the element to the caller. Otherwise, it must try to obtain a P-lock for a *Deq/Failed* event, and the whole process starts over again. (In its current form, this server is unfair to *Deq* operations since *Enq* and *Eval* events could lock them out indefinitely. This situation could be avoided by assigning a priority to the waiting *Deq* operation.)

An *Eval* operation invocation is handled similarly to the *Enq* operation. When a transaction invokes an *Eval* operation, the server first checks to ensure that no other active transaction holds a P-lock on either a *Deq/Ok(i)* or an *Enq(i)/Ok* event. If no such conflicts exist, then it assigns the calling transaction a P-lock for an Eval event, and returns a total to the caller (this total takes into account the calling transaction's intentions list). However, if there is a conflict, the Eval invocation will block until it receives a wake-up message.

When a transaction has successfully obtained all of its neccesary P-locks and has finished executing, it inflicts the changes recorded in its *Enq* and *Deq* intentions list to the global state of the semiqueue by invoking an atomic *Global_Update* operation. Once these changes have been accomplished, the committing transaction's P-locks can be removed and wake-up calls made to any blocked

invocations that could benefit by these releases.

### 4.2.2.1 Deadlock Detection

As previously discussed (section 3.2.2), the pessimistic server includes a mechanism (operation *Deadlock_Check* ) which can prevent deadlock between two transactions. Specifically, this mechanism considers the following five cases:

Case I: ABORT(Ti) if Ti is about to block on an *Enq(item)/Ok* event and there exists a Tj such that:
(i) Tj holds a *Deq()/Failed* P-lock and
(ii) Tj is waiting for the release of a lock-type that Ti holds.

Case II: ABORT(Ti) if Ti is about to block on a *Deq()/Ok(item)* event and there exists a Tj such that:
(i) Tj holds a *Deq()/Ok* P-lock and
(ii) Tj is waiting for the release of a lock-type that Ti holds.

Case III: ABORT(Ti) if Ti is about to block on a *Deq/Failed* event and there exists a Tj such that:
(i) Tj holds an *Enq/Ok* P-lock and
(ii) Tj is waiting for the release of a lock-type that Ti holds.

Case IV: ABORT(Ti) if Ti is about to block on an *Eval/(#items)* event and there exists a Tj such that:
(i) Tj holds an *Enq/Ok* or a *Deq/Ok* P-lock and
(ii) Tj is waiting for the release of a lock-type that Ti holds.

Case V: ABORT(Ti) if Ti is about to block on an *Enq/Ok* or a *Deq/Ok* event and there exists a Tj such that:
(i) Tj holds an *Eval/(#items)* P-lock and
(ii) Tj is waiting for the release of a lock-type that Ti holds.

### 4.2.2.2 Wake-Up Messages

Wake-up messages are issued to all blocked operations whenever a transaction releases a P-lock that could potentially unblock a waiting operation. P-locks are released whenever a transaction commits, aborts, or dequeues the last element in its enqueue intentions list (section 4.2.2).

When a transaction commits (i.e., enters the *Global_Update* procedure), its enqueue intentions list is appended to the end of the semiqueue, the shared elements that are referenced by the dequeue intentions list are removed, and all P-locks are released. Before the server becomes available to handle new operation requests, however, wake-up calls are issued to all the appropriate blocked operations after the committing transaction's P-locks have been released. Given that *Tnum* is the transaction number of the committing transaction and *N* is the maximum number of active transactions supported by the server, wake-up calls are performed in the following manner (in accordance to dependencies D1 - D7 described in section 3.2):

```
        if Tnum held an Enq P-lock ->
           for all transaction numbers t from 1 to N (other than Tnum) ->
(D1)             if t is blocked in a Deq/Failed event ->
                    issue a Wake-up call to the operation
                    and wait for a response
                 fi
           af
        fi
```

```
            if Tnum held a Deq/Failed or Eval P-lock ->
               for all transaction numbers t from 1 to N (other than Tnum) ->
(D5,D6)                  if t is blocked in an Enq/Ok event ->
                            issue a Wake-up call to the operation
                            and wait for a response
                         fi
               af
            fi


            if Tnum held an Eval P-lock ->
               for all transaction numbers t from 1 to N (other than Tnum) ->
(D7)                     if t is blocked in a Deq/Ok(i) event ->
                            issue a Wake-up call to the operation
                            and wait for a response
                         fi
               af
            fi


            if Tnum held an Enq P-lock or Deq P-lock
            (for any shared semiQ item) ->
               for all transaction numbers t from 1 to N (other than Tnum) ->
(D2, D4)                 if t is blocked in a Eval/(#items) event ->
                            issue a Wake-up call to the operation
                            and wait for a response
                         fi
(D3)                     if t is blocked in a Deq invocation ->
                            issue a Wake-up call to the operation
                            and wait for a response
                         fi
               af
            fi
```

When a blocked operation receives a wake-up call, it first checks to see if it can be unblocked. In the case where there has been a release of the appropriate P-lock type and the operation can proceed, it is performed at this time and control is

returned to the issuer of the wake-up call once its execution is complete. If, however, it is still blocked by other P-locks, it merely reblocks and control is returned to the issuer of the wake-up call.

The *Abort* procedure issues essentially the same wake-up calls as those listed for for the *Global_Update* procedure, but it must handle D3 a little differently. A waiting *Deq* invocation could potentially be unblocked by a committing transaction that has either enqueued or dequeued elements. In the case where a committing transaction has enqueued a new element, *i*, this element becomes eligible and the blocked *Deq* invocation could subsequently perform a *Deq/Ok(i)* event. Or, in the case where all shared elements that are not already Deq P-locked by the blocked transaction are removed by the committing transaction, the blocked invocation could subsequently perform a *Deq/Failed* event. When a transaction is being aborted, however, the release of an Enq P-lock does not coincide with the addition of eligible elements to the semiqueue. Similarly, the release of Deq P-locks does not result in the removal of elements. Instead, the release of Deq-Plocks associated with shared elements of the semiqueue by an aborting transaction makes those elements eligible for dequeuing once more. Hence, wake-up calls (for D3) are only given when the aborting transaction's *Deq* intentions list is thrown out, and not issued upon release of Enq P-locks.

Special attention must be paid to dequeue operations which access items that are local to a transaction's enqueue intentions list since they constitute a special case

where an Enq P-lock could be released. When a transaction's dequeue operation removes the only item that is local to its enqueue intentions list, it is no longer necessary for that transaction to hold an Enq P-lock. Since the release of this lock could potentially unblock any Deq/Failed events that have been forced to wait, wake-up calls (in accordance to D1) are issued at this time, before the server becomes available for new operation invocations.

### 4.2.3 Hybrid Operations

The hybrid server combines the boolean arrays of O-locks from the optimistic server (Figure 4.5) with the semiqueue node-types of the pessimistic server (Figure 4.8). As previously established (section 3.3), the four proscribed dependency relations employed by the hybrid method are treated as follows:

**D1**: $T_i$:Enq(i)/Ok() ->$_q$ $T_j$:Deq()/Failed() (Optimistic)
**D2**: $T_i$:Enq(i)/Ok() ->$_q$ $T_j$:Eval()/Ok(#items) (Optimistic)
**D3**: $T_i$:Deq()/Ok(i) ->$_q$ $T_j$:Deq()/Ok(i) (Pessimistic)
**D4**: $T_i$:Deq(i)/Ok() ->$_q$ $T_j$:Eval()/Ok(#items) (Optimistic)

The implementation of the *Enq* and *Eval* operations in this server are identical to those described for the optimistic server (section 4.2.1), and a *Deq* operation closely resembles the pessimistic implementation.

As with the other servers, a *Deq* operation is invoked with the *Tnum* of the calling transaction and begins with a check of that Tnum's local *Enq* intentions list. If this list contains at least one element, the *Deq* operation removes this element and returns its value (unlike the pessimistic server, there are no locks associated with this server's optimistic *Enq* operation), otherwise the shared semiqueue is accessed. If the semiqueue is either empty or all elements are already P-locked by the calling transaction (i.e., there are no eligible elements), a *failed* result is returned and an O-lock is allocated for this *Deq/Failed* event. If, however, all eligible elements are P-locked by other transactions, the *Deq* invocation is blocked (according to proscribed dependency D3 above). As with the pessimistic server, wake-up calls are issued from any validating transaction that has enqueued at least one item or any aborting transaction that was attempting to dequeue a shared element. When awoken, this invocation re-examines the shared queue for eligible elements, and this process starts again.

Since the hybrid server only treats one type of conflict pessimistically, the *Deadlock-Check* operation is much less substantial than that of the pessimistic server. Instead of considering the five cases outlined in section 4.2.2, only one case needs to be dealt with:

Case II: ABORT(Ti) if Ti is about to block on a *Deq()/Ok(item)* event and there exists a Tj such that:
(i) Tj holds a *Deq()/Ok* P-lock and
(ii) Tj is waiting for the release of a lock-type that Ti holds.

Due to the fact that the only blocking lock-type is a *Deq/Ok* P-lock, deadlock could arise only after two active transactions have both successfully dequeued items from the shared semiqueue, and then they both block on subsequent *Deq* invocations. The *Abort* operation essentially remains the same as in the pessimistic and optimistic servers; intentions lists are thrown out and all locks are released.

In order to accommodate both the optimistic and pessimistic treatment of conflicts, the hybrid server includes both the*Validation* and *Global_Update* operations. As within the optimistic server, validation (section 4.2.1) proceeds through step one and a modified version of step two (which excludes an O-lock check for Deq/Ok events since they are handled pessimistically):

```
Step Two:   if Deq Intentions list[Tnum] ≠ ø ->
                for all transaction numbers, i, ≠ Tnum ->
                    if  Eval_Olock[i]  ->
                        ABORT(Tnum) and return
                    fi
                af
            fi
```

The *Global_Update* operation inflicts the changes recorded in a transaction's *Enq* and *Deq* intentions list onto the global state of the semiqueue, at which point all locks can be removed and the appropriate wake-up calls can be issued.

# 5 Tests and Results

In an effort to establish some relative performance characteristics, we subjected each of the pessimistic, optimistic and hybrid semiqueue servers to four sets of tests. The purpose of each test set was to determine the servers' behavior under increasing levels of type-specific conflict. The particular conflict types tested are the four proscribed serial dependencies originally defined for the optimistic server (section 3.1), which are also included in the proscribed dependency sets for each of the pessimistic and hybrid approaches.

In order to evaluate the respective performances of each of the three approaches fairly, all tests were carefully contrived to accomplish the same amount of "work" at each server (i.e., within a given level of concurrency, each server must successfully execute the same number and type of events). Operation requests from different transactions are simulated by invoking a server with unique transaction identification numbers used to represent active transactions in the system. Each of the servers is capable of supporting up to 100 concurrently active transactions at any given time, with transaction identification numbers ranging from *Trans(1)* to *Trans(100)*.

## 5.1 Testing Enq(i)/Ok --> Deq()/Failed and Enq(i)/Ok --> Eval/(#items) Conflict Types

The first conflict type tested among the servers deals with the proscribed dependency D1:

**D1:** $T_i$: Enq(i)/Ok() -->$_q$ $T_j$: Deq()/Failed()

In this set of tests, each server attempts to handle its maximum of 100 concurrently active transactions, and each transaction must successfully enqueue (that is, perform *Enq(i)/Ok* events) 100 elements. Conflict is introduced as a result of the fact that at the very beginning of the test, one of the transactions, Trans(*Tnum* ), performs a *Deq/Failed* event. The percentage of conflict is then determined by the number of transactions blocked or aborted due to their conflict with the active transaction Trans(*Tnum* ). This percentage of conflict is controlled by running the tests with different values of *Tnum* (where the actual percentage of conflict would be *Tnum*-1 in each test[4]). The pessimistic, optimistic and hybrid tests for this type of conflict were devised as follows:

Pessimistic *Enq(i)/Ok* --> *Deq/Failed* Test:

(i) Trans(*Tnum* ) performs an *Deq/Failed* event and then performs *Enq(i)/Ok* events for 100 elements.

---

[4] Our tests results were based on *Tnum* values of 1, 31, 61 and 91 to arrange 0%, 30%, 60% and 90% conflict levels, respectively. The results included in this performance analysis represent the best times obtained from multiple tests run when the system load was light, but not negligible.

(ii) Trans(1) through Trans(*Tnum* -1) each attempt to *Enq* an element, but all are blocked by Trans(*Tnum* ).

(iii) Trans(*Tnum* ) performs a *Global_Update* , unblocking Trans(1) through Trans(*Tnum* -1), which each then complete their initial *Enq(i)/Ok* events after receiving their respective wake-up messages.

(iv) Trans(1) through Trans(*Tnum* -1) execute *Enq(i)/Ok* events for 99 more elements each.

(v) Trans(*Tnum* +1) through Trans(100) execute *Enq(i)/Ok* events for 100 elements each.

(vi) All transactions except Trans(*Tnum* ) perform *Global_Update.*

Optimistic *Enq(i)/Ok --> Deq/Failed* Test:

(i) Trans(*Tnum* ) performs an *Deq/Failed* event.

(ii) Trans(1) through Trans(100) each perform *Enq(i)/Ok* events for 100 elements.

(iii) Trans(1) through Trans(*Tnum* ) attempt *Validation* (in that order), but only Trans(*Tnum* ) goes on to its *Write* phase, the rest are aborted as a result of the conflict with Trans(*Tnum* ).

(iv) Trans(1) through Trans(*Tnum* -1) are redone.

(v) All transactions except Trans(*Tnum* ) complete their respective *Validation* and *Write* phases.

Hybrid *Enq(i)/Ok --> Deq/Failed* Test:

(i) Trans(*Tnum* ) performs an *Deq/Failed* event.

(ii) Trans(1) through Trans(100) perform *Enq(i)/Ok* events for 100 elements each.

(iii) Trans(1) through Trans(*Tnum* ) attempt *Validation* , but only Trans(*Tnum* ) goes on to execute *Global_Update* , the rest are

aborted.

(iv) Trans(1) through Trans(*Tnum* -1) are redone.

(v) All transactions except Trans(*Tnum* ) successfully perform *Validation* and subsequently proceed to *Global_Update* .

Due to the optimistic nature of the hybrid server's handling of this type of conflict, the hybrid test differs from the optimisitic test only in the inherent fact that optimistic*Validation* includes a *Write* phase, whereas hybrid *Validation* relies on a seperate *Global_Update* procedure (steps (iii) and (v)).

Identical steps were followed to contrive the second set of tests, which deals with the proscribed dependency D2:

**D2:** $T_i$: Enq(i)/Ok() -->$_q$ $T_j$: Eval/(#items)

In these tests, *Eval/(#items)* is substituted for *Deq/Failed* as the event performed by Trans(*Tnum* ) to introduce conflict. Not surprisingly, (since Trans(*Tnum* ) executes *Eval(#items)* on a an empty semiqueue) these test results are indistinguishable from those produced by the tests of conflict type D1 (Figure 5.1).

### 5.1.1 Results

The respective performance of the semiqueue servers as they are subjected to an increasing percentage of *Enq(i)/Ok -> Deq/Failed* and *Enq(i)/Ok -> Eval/(#items)*

conflict type is as follows (where performance times are reported to the nearest second):

Time (in secs) for 100 transactions to perform 100 Enq(i)/Ok events and/or one Deq/Failed, Eval/(#items) event.

| Server Type: | | Optimistic | Pessimistic | Hybrid |
|---|---|---|---|---|
| Conflict Level: | 0% | 11 | 16 | 5 |
| | 30% | 14 | 16 | 7 |
| | 60% | 18 | 16 | 10 |
| | 90% | 21 | 17 | 12 |

Which can roughly be depicted as:

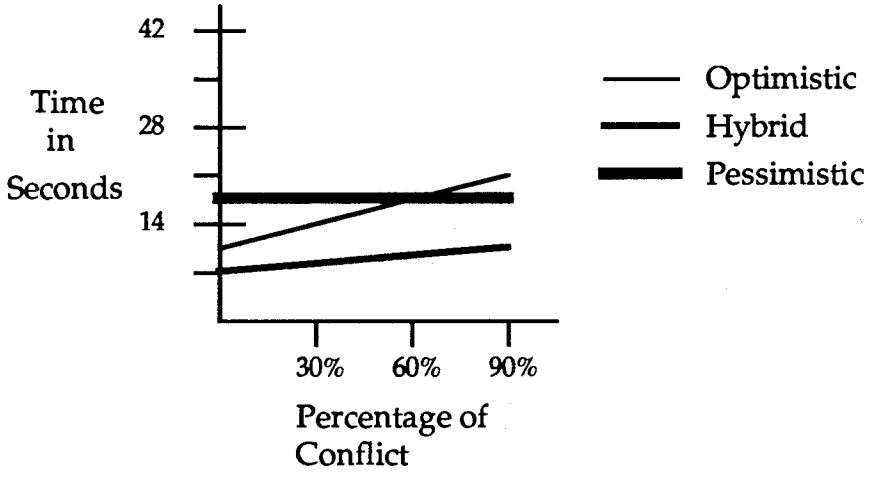



**Figure 5.1**: *Enq(i)/Ok -> Deq/Failed* and *Enq(i)/Ok -> Eval/(#items)* conflicts

These results indicate that when subjected to an increasing level of the type specific conflict defined by dependencies D1 and D2, the pessimistic server's performance remains virtually unaffected, the optimistic server's performance steadily deteriorates and the hybrid server out performs the other servers throughout the set of tests.

In the pessimistic server's case, the fact that these tests are heavily work

intensive (i.e., 100 concurrent transactions enqueuing 100 elements each) makes the additional cost of *Wake-up* calls that accompany an increasing level of transaction conflict essentially negligable. That is, since the number and type of events executed at each level of conflict is the same, its performance is basically unchanging. In comparison to the other servers, when there are no conflicts among transactions the pessimistic server is the most expensive. This can be attributed to the fact that the pessimistic *Enq(i)/Ok* event requires more locking overhead than it does in the other servers.

With the optimistic server, the amount of work that must be redone is directly proportional to the percentage of conflict in each test. That is, at 30%, 60% and 90% conflict, 30, 60 and 90 of the 100 transactions must be redone. Consequently, since the amount of work steadily increases as the percentage of conflict increases, the amount of time required to process the work inceases, forcing its performance to steadily deteriorate. It is interesting to note that the threshold value where the optimistic and pessimistic servers' performance intersect is at approximately 50% for these particular types of conflict.

In the environment created by these tests, the performance of the hybrid server is superior to both the optimistic and pessimistic servers at any given percentage of conflict. Although the hybrid server treats the conflict types being tested optimistically (consequently requiring that more work must be done as the percentage of conflict increases), the affect on performance is not as significant for

the hybrid server as it is for the optimistic server. This is due to the fact that the *Enq(i)/Ok* event is less expensive in the hybrid server since the nodes of the hybrid semiqueue require only one P-lock for its pessimistic *Deq()/Ok(i)* event (section 4.2.3, Figure 4.8), whereas the optimistic semiqueue requires N+1 (where N the maximum number of concurrent transactions, i.e., 100 in this case) O-locks in each node (section 4.2.1, Figure 4.4).

## 5.2 Testing the Deq()/Ok(i) --> Deq()/Ok(i) Conflict Type

The third set of tests deals with the proscribed dependency D3:

**D3:** $T_i$: Deq()/Ok(i) $\rightarrow_q$ $T_j$: Deq()/Ok(i).

Within these tests, each server is again attempting to handle its maximum of 100 concurrently executing transactions, of which we determine that 99 transactions must successfully dequeue (that is, perform *Deq()/Ok(i)* events) 30 elements each. In order to simulate this type of conflict, the semiqueue is initially set up with 2970 shared elements. To establish a controlled percentage of conflict similar to the first two sets of tests, transaction Trans(1) acts as a "dummy" transaction and holds *Deq()/Ok(i)* locks on ((*Tnum* -1)*30) of these elements. During the test, once the desired level of (*Tnum* - 1)% conflict has been achieved, Trans(1) is forcefully aborted (just for testing purposes), making its previously locked elements once again

eligible for dequeueing by other transactions. These tests proceed as follows:

Pessimistic *Deq()/Ok(i)* --> *Deq()/Ok(i)* Test:

(Trans(1) performs a *Deq/Ok(i)* event on (Tnum-1)*30 elements -- this is not included in the timing results of the test.)

(i) Trans(2) through Trans(101-*Tnum* ) perform *Deq/Ok(i)* for 30 elements each.

(ii) Trans(102-*Tnum* ) through Trans(100) each attempt to *Deq* one element, but all are blocked.

(iii) Trans(1) is manually aborted (for test purposes only), and blocked transactions complete their initial *Deq/Ok(i)* events.

(iv) Trans(102-*Tnum* ) through Trans(100) each perform 29 more *Deq/Ok(i)* events.

(v) All transactions except Tnum(1) perform *Global_Update*.

Optimistic *Deq()/Ok(i)* --> *Deq()/Ok(i)* Test:

(Trans(1) performs a *Deq/Ok(i)* event on (Tnum-1)*30 elements -- this is not included in the timing results of the test.)

(i) Trans(2) through Trans(100) each perform *Deq/Ok(i)* events for 30 elements.

(ii) Trans(102-*Tnum* ) through Trans(100) attempt *Validation* , but all abort.

(iii) Trans(1) is manually aborted (for test purposes only).

(iv) Trans(102-*Tnum* ) through Trans(100) are redone.

(v) All transactions except Trans(1) successfully perform *Validation* and their subsequent *Write* phases.

Due to the pessimistic nature of this type of conflict in the hybrid server, the hybrid test for this type of conflict is identical to that of the pessimistic server's, except for the inclusion of *Validation* (in step (v)):

Hybrid *Deq()/Ok(i)* --> *Deq()/Ok(i)* Test:

(Trans(1) performs a *Deq/Ok(i)* event on (Tnum-1)*30 elements -- this is not included in the timing results of the test.)

(i) Trans(2) through Trans(101-*Tnum* ) perform *Deq/Ok(i)* for 30 elements each.

(ii) Trans(102-*Tnum* ) through Trans(100) each attempt to *Deq* one element, but all are blocked.

(iii) Trans(1) is manually aborted (for test purposes only), and blocked transactions complete their initial *Deq/Ok(i)* events.

(iv) Trans(102-*Tnum* ) through Trans(100) each perform 29 more *Deq/Ok(i)* events.

(v) All transactions except Tnum(1) successfully perform *Validation* and *Global_Update.*

### 5.2.1 Results

Time (in secs) for 99 trans to Deq 30 items each

| Server: | Opt | Pess | Hyb |
|---|---|---|---|
| Conflict: 0% | 39 | 58 | 58 |
| 30% | 64 | 59 | 59 |
| 60% | 87 | 59 | 60 |
| 90% | 111 | 60 | 60 |

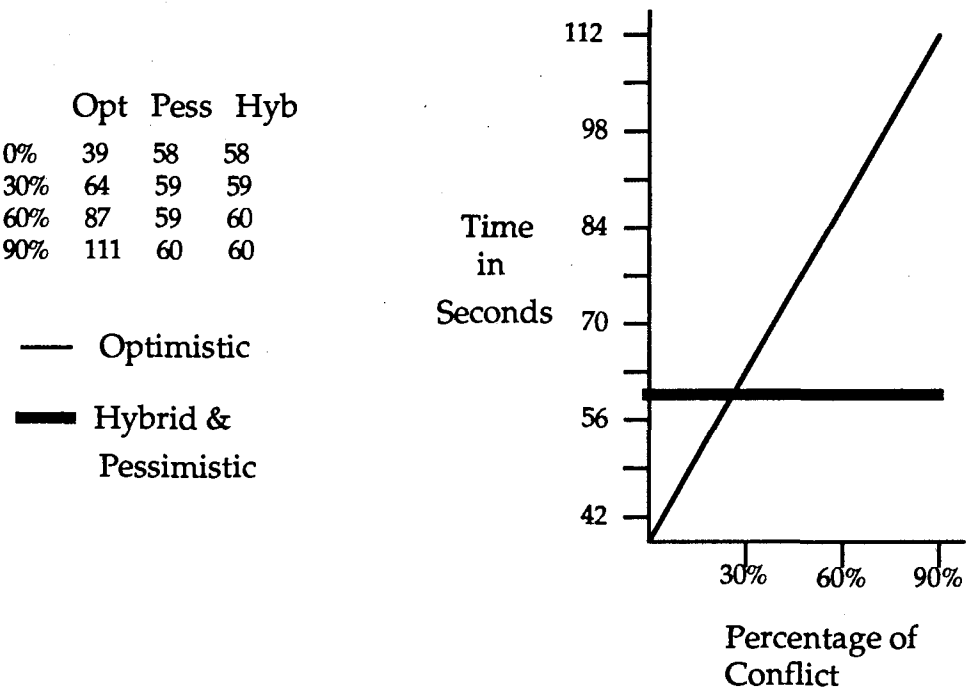



**Figure 5.2:** *Deq/Ok(i) -> Deq/Ok(i)* conflicts

As with the previous results obtained by testing the type specific conflicts defined by dependencies D1 and D2, within these tests the pessimistic server's performance remains essentially unaffected by an increasing percentage of conflict and the optimistic server's performance deteriorates proportionally. Due to the pessimistic nature of the hybrid server's handling of this type of conflict, the hybrid server's performance is virtually equivalent to that of the pessimistic server's. It is interesting to note that the percentage of conflict representing the threshold between

optimistic and pessimistic performance falls below 30% for this type of conflict.

## 5.3 Testing the Deq/Ok(i) --> Eval/(#items) Conflict Type

The final set of tests deals with the proscribed serial dependency D4:

**D4:** $T_i$: Deq()/Ok(i) $-->_q$ $T_j$: Eval/(#items).

With these tests, the semiqueue initially has 3000 items, and each of the 100 transactions must dequeue 30 elements. In addition to performing these events, Trans(*Tnum* ) also performs an *Eval/(#items)* event (as in section 5.1.1) on the initial queue. The tests proceed as follows:

Pessimistic Deq/Ok(i) --> Eval/(#items) test:

(i) Trans(*Tnum* ) performs an *Eval/(#items)* event and 30 *Deq/Ok(i)* events.

(ii) Trans(1) through Trans(*Tnum* -1) attempt to *Deq* one element each, and all are blocked on Trans(*Tnum* ).

(iii) Trans(*Tnum* ) performs a *Global_Update* , all blocked transactions are subsequently woken up and able to complete their initial *Deq/Ok(i)* events.

(iv) Trans(1) through Trans(*Tnum* -1) each perform 29 more *Deq/Ok(i)* events.

(v) Trans(*Tnum* +1) through Trans(100) each perform 30 *Deq/Ok(i)* events.

(vi) All transactions except Trans(*Tnum* ) perform a *Global_Update*.

Optimistic Deq/Ok(i) --> Eval/(#items) test:

(i) Trans(*Tnum* ) performs an *Eval/#items* event.

(ii) Trans(1) through Trans(100) each perform 30 *Deq/Ok(i)* events.

(iii) Trans(1) through Trans(*Tnum* ) perform *Validation* , but only Trans(*Tnum* ) proceeds to its *Write* phase, the rest are aborted.

(iv) Trans(1) through Trans(*Tnum* -1) are redone.

(v) All transactions except Trans(*Tnum* ) perform *Validation* and go on to their respective *Write* phases.

Once again, due to the optimistic nature of this type of conflict in the hybrid server, the hybrid test is identical to that of the optimistic server except for the fact that *Global_Update* is a separate invocation.

### 5.3.1 Results

The results in Figure 5.3 indicate, when subjected to an increasing level of the type specific conflict defined by dependency D4, the pessimistic server's performance is still virtually unaffected and the optimistic server's performance deteriorates. This time the hybrid server is out performed by the others throughout this set of tests.

The performance of the pessimistic server under these test conditions is essentially the same as its performance in the previous set of tests for the *Deq/Ok(i) ->Deq/Ok(i)* type of conflict. This indicates that the additional events performed in these tests for the *Deq/Ok(i)-> Eval/(#items)* type of conflict (which are an *Eval/(3000)* event along with 30 more *Deq/Ok(i)* events because all 100 transactions are dequeuing as opposed to 99 in the previous tests) are insignificant in the pessimistic context. Once again, its performance is unaffected by an increasing percentage of conflict due to the fact that the same number and type of events are being processed in each case.

Time (in secs) for 100 trans to Deq 30 elements each
and perform one Eval/(#items) event

| Server: | Opt | Pess | Hyb |
|---|---|---|---|
| Conflict: 0% | 42 | 59 | 59 |
| 30% | 47 | 59 | 70 |
| 60% | 58 | 59 | 87 |
| 90% | 73 | 59 | 107 |

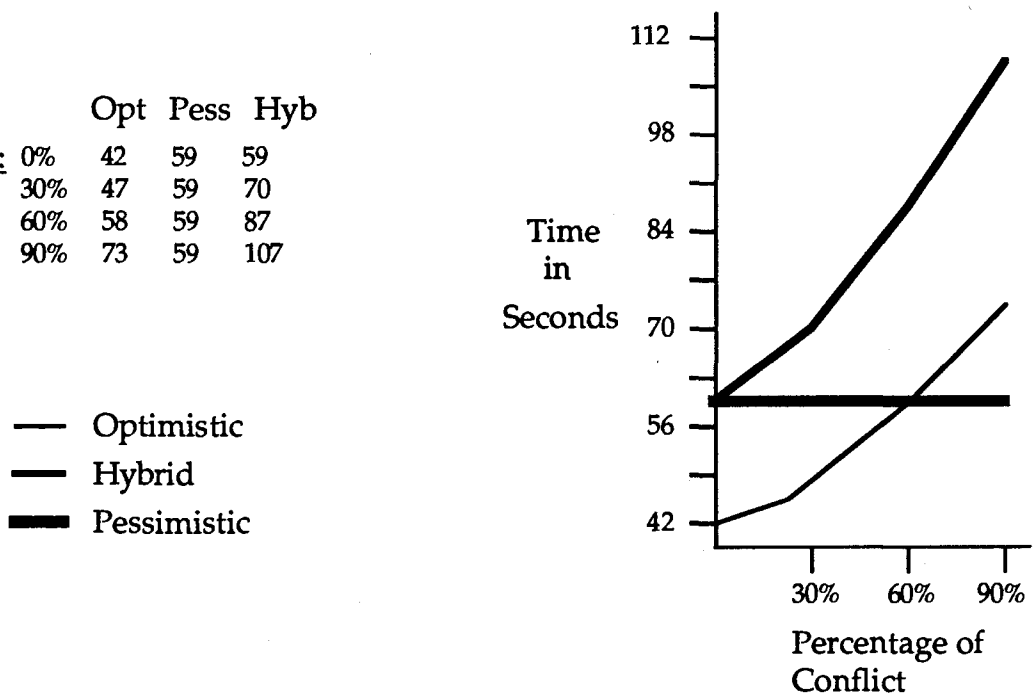



**Figure 5.3:** *Deq/Ok(i) --> Eval/(#items)* conflicts

A comparison of these performance results with those of the previous set of tests (Figure 5.2) indicates that the optimistic server is more sensitive than the robust pessimistic server to the additional performance costs introduced by the *Eval/(3000)* event and the 100th transaction's execution of 30 *Deq/Ok(i)* events (which are the only difference between the tests in Figure 5.2 and Figure 5.3). This is demonstrated by the discrepancy between the performance of the optimistic server at 0% conflict in Figure 5.2 and Figure 5.3 (39 secs vs 42 secs), whereas the performance of the pessimistic server was virtually unaffected (58 secs vs 59 secs). Almost immediately, however, as the percentage of conflict increases, these initial costs are absorbed by the gains associated with the elimination of unsuccessful searches for eligible elements. That is, in the tests for conflicts defined by D3, before a transaction can set its Deq O-lock on an element that is already O-locked by another transaction, it must first search the entire semiqueue for an unlocked element. Since these fruitless searches do not occur under the test conditions for conflicts defined by D4, the *Deq/Ok(i)* events are less expensive and consequently the costs of redoing them are less. This is demonstrated by the discrepancy between the performance of the optimistic server at 90% in Figure 5.2 and Figure 5.3 (111 secs vs 73 secs). It is interesting to note that under these conditions the threshold value where the performance results for the pessimistic and optimistic servers intersect is at approximately 60%.

In the case of the hybrid server, its performance is equivalent to the pessimistic server at 0% conflict, then deteriorates as the percentage of conflict increases. This is

due to the fact that the cost of the hybrid *Deq/Ok(i)* event is largely determined by the overhead associated with the pessimistic treatment of the conflict type defined by D3, whereas the handling of the *Deq/Ok(i) -> Eval/(#items)* conflict type defined by D4 is optimistic. This means that an increasing number of relatively expensive *Deq/Ok(i)* events must be redone as the percentage of conflict increases, which results in an even more dramatic deterioration of performance than the optimistic server.

# 6 Evaluation and Extension

## 6.1 Relative Behavior

Clearly, the tests performed in our study are representative of only a very small sample of the type of transaction that could potentially access this shared abstract object. A "typical" extended transaction defies definition however, due to the fact that number and type of events executed by each transaction is dependent on the application from which it is issued. Consequently, instead of attempting to provide a representative sampling of extended transactions, the intent of these tests is to establish some of the fundamental characteristics that determine the relative behavior of the three different concurrency control methods. Identifying these characteristics can serve to better define the environments in which each approach is most effective.

### 6.1.1 Pessimistic Behavior

One of the most fundamental characteristics inherent in any pessimistic locking method is the fact that it does not allow for any wasted work. As a result, within the environment of our "work intensive" tests, the pessimitic server's

performance is essentially unaffected by the increasing level of transaction conflict. Another beneficial aspect of type-specific locking is that deadlock prevention and the method by which transactions are unblocked can be customized through the exploitation of semantic information (section 4.2.2.1 and section 4.2.2.2).

Of course, as with any pessimistic locking scheme, the major disadvantage associated with this method is the fact that the inherent overhead incurred by lock management and deadlock prevention (or detection) is superfluous when the level of conflict is sufficiently low. The effect of this locking overhead in our servers can be demonstrated by comparing the pessimistic and optimistic performance results at 0% conflict (Figure 5.1: 16 vs ll secs, Figure 5.2: 58 vs 39 secs, and Figure 5.3: 59 vs 42 secs). In an environment where the percentage of conflict is sufficiently low, the additional overhead associated with lock management in the pessimistic case can be be partially attributed to the fact that the proscribed dependencies used to define type-specific conflicts must be symmetrical in a pessimistic scheme, and hence it has more conflict types than its optimistic counter-part.

Further additional overhead is inherent in our implementation of the pessimistic technique due to its manditory check for conflicts after the execution of each event. For example, each time an element is to be enqueued in the pessimistic approach, a conflict check for *Deq/Failed* or *Eval* P-locks must be performed before updating the calling transaction's Enq intentions list. In the optimistic approach however, this type of conflict check is executed at most once during the validation

phase of a given transaction, regardless of how many *Enq(i)/Ok* events that transaction performed.

### 6.1.2 Optimistic Behavior

The major advantages associated with the optimistic approach directly correspond to the disadvantages mentioned in the context of the pessimistic method. That is, there is an absence of the overhead inherent in locking management and deadlock prevention, there are fewer conflicts to be considered due to the fact that the proscribed dependency relations are not symmetrical, and conflict detection can be cheaper when it does not have to be on a per event basis. A further advantage of this particular optimistic approach is its ability to sustain an impressive level of performance relative to its pessimistic counter-part up until a substantial level of transaction conflict has been established. Taking the average threshold value of the four sets of tests we ran (which represent the only types of conflict encountered by the optimistic approach), the optimistic server is able to perform better than or at least as well as the pessimistic server up until the level of conflict rises above approximately 45%.

One of the major disadvantages inherent in all optimistic approaches is the fact that under increasing levels of conflict, the problem of wasted work intensifies.

The set of tests associated with the type specific conflict defined by D3: *Deq/Ok(i)->* *Deq/Ok(i)* , (Figure 5.2) best demonstrates the dramatic contrast in the amount of wasted work performed by the pessimistic and optimistic servers. In the pessimistic version of this test at 90% conflict, once all of the shared elements have been P-locked the pessimistic server attempts to execute one *Deq/Ok(i)* event for each of the 90 different transactions involved in the conflict. Every *Deq* invocation searches the semiqueue and then becomes blocked until the dummy transaction, Trans(1), is aborted. In contrast, the optimistic server executes 30 *Deq/Ok(i)* events for each of the 90 different transactions, all of which must be redone. Furthermore, each of these optimistic *Deq* operations search the entire semiqueue for an eligible element before setting an O-lock on an element that is already locked by another transaction. The fact that a single element (node) in the semiqueue could potentially be O-locked by all the concurrent transactions in the system requires that O-lock information for each transaction to be associated with every element of the semiqueue. The fact that nodes in the hybrid server do not include this extensive O-lock information (section 6.1.3) and that its performance in the first two sets of tests (Figure 5.1) is notably superior to the optimistic server indicates that these locks can be burdensome.

### 6.1.3 Hybrid Behavior

The hybrid server was designed (section 3.3) to be appropriate under conditions where concurrent *Deq/Ok(i)* operations are expected to be frequent (hence, conflicts of type D3 are treated pessimistically) but all other conflict types are expected to be rare (and therefore are treated optimistically).

In theory, the advantage of the hybrid methodology we used (as presented in [Herlihy86] ) is that it enables the appropriate optimistic or pessimistic concurrency control method to be applied on a per conflict-type basis according to the associated probability of the conflict. As demonstrated by the first two sets of tests in our study (Figure 5.1), this method can exploit the combined individual strengths of both the optimistic and pessimistic approaches and subsequently outperform both methods under certain conditions. The strength provided by the optimistic component of the hybrid server is the low overhead costs associated with its fewer type-specific conflicts and its minimal needs for pessimistic lock management and deadlock detection. The strength provided by the hybrid server's pessimistic component is the fact that the extensive O-lock information associated with each element of the semiqueue is simply replaced by one P-lock. Consequently, the hybrid server's performance is superior to that of the optimistic server in our first two sets of tests (Figure 5.1). Equally as encouraging are the results from the third set of tests (Figure 5.2) which demonstrate the hybrid server's ability to handle pessimistic conflict types

at a performance level that is on par with the pessimistic server's performance.

The major disadvantage associated with the hybrid approach is that although this method can combine the individual strengths of both its optimistic and pessimistic components, it can also combine their weaknesses. This is most obviously demonstrated within the context of the fourth set of tests we performed (Figure 5.3). These tests study D4, the *Deq/Ok(i)->Eval/(#items)* conflict type. In the hybrid server, *Deq/Ok(i)* events are inherently expensive due to the locking and deadlock prevention overhead associated with its pessimistically treated conflict *Deq/Ok(i)->Deq/Ok(i)*. This expense can be considered to be the weakness contributed to the hybrid approach by its pessimistic component. The weakness of its optimistic component is its inherent increase in the amount of wasted work under increasing percentages of conflict. In the hybrid test for D4, the relatively expensive *Deq/Ok(i)* events are involved in a conflict that is treated optimistically. Hence this lavish operation is repeatedly redone and the resulting performance of the hybrid server becomes lamentable as the percentage of conflict increases. Since hybrid performance is not just dependent on the level of transaction conflict but also contingent on the particular type of conflict encountered, this approach is only appropriate within very narrow and controlled conditions. Involving a hybrid operation that includes abundant pessimistic overhead in an optimistically treated conflict type will inevitably lead to atrocious performance relative to pure optimistic or pessimistic behavior.

### 6.1.4 Summary of Behavior

The major advantages, disadvantages, and appropriate environment for each of the pessimistic, optimistic and hybrid approaches studied here can be summarized in the following way:

| | ADVANTAGES | DISADVANTAGES | ENVIRONMENT |
|---|---|---|---|
| PURE PESSIMISTIC | 1. No wasted work.<br>2. Unaffected by conflicts.<br>3. Customizable deadlock detection & unblocking. | 1. High overhead at low % of conflict.<br>2. Symmetricial conflicts.<br>3. Conflict detection check after every event. | Best suited for situations where expected level of all conflicts > 45% |
| PURE OPTIMISTIC | 1. Absence of pessimistic overhead (lock/deadlock)<br>2. Fewer type-specific conflicts. | 1. Much wasted work at higher levels of conflict.<br>2. Extensive O-locking can introduce overhead. | Best suited for situations where expected level of all conflicts < 45% |
| HYBRID | 1. Offers opt/pess choice according to associated probability of conflict.<br>2. Can exploit combined strengths. | 2. Can combine weaknesses of both methods when pessimistic overhead is in an optimistic conflict. | Best suited for situations where % of conflict varies with each conflict-type. |

**Figure 6.1**: Fundamental Characteristics of Pessimistic, Optimistic and Hybrid Techniques

## 6.2 Extension: Directories

The effectiveness of the customizable type-specific method of concurrency control studied here can be further exemplified by its application to another abstract object. Consider a *directory* object which supports the following types of events[5]:

Insert(str, capa)/Ok() -- inserts capability *capa* into with key string *str*.
Insert(str, capa)/Failed() -- a failed insert event due to duplicate key string.
Delete(str)/Ok() -- deletes capa stored with *str* .
Delete(str)/Failed() -- a failed delete event due to *str* not found.
Lookup( str)/Ok(capa) -- searches for a capa with key *str*.
Lookup(str)/Failed() -- a failed lookup event due to *str* not found.
Eval()/Ok(#entries) -- returns the number of entries in the directory.

The use of semantic information associated with the type-specific directory operations can greatly increase the amount of concurrency allowed by a synchronization scheme. For example, if a conventional READ/WRITE locking scheme was used to control these operations, *Insert* and *Delete* could each be treated as a READ operation followed by a WRITE operation, whereas *Lookup* and *Eval* could each be treated as READs. In terms of pessimistic concurrency restrictions, a successful *Lookup("foo")* invocation would be blocked trying to obtain a READ lock if another transaction performed a successful *Delete("fum")*

---

5 An event consists of an operation and a response.

event, thus holding a WRITE lock on the directory. This restriction on concurrency is unnecessary, and can be alleviated through the use of a type-specific approach.

The number of proscribed serial dependencies needed to define type-specific conflicts can be minimized by dividing the seven possible events associated with the directory data type into three lock classes [Schwarz84]:

M = for events that "Modify" a particular entry:
*Insert(str, capa)/Ok(), Delete(str)/Ok()* .

L = for events that only "Look" at a particular entry:
*Lookup(str)/Ok(), Lookup(str)/Failed(),*
*Insert(str, capa)/Failed(), Delete(str)/Failed().*

E = for events that cannot be isolated to a particular entry:
*Eval()/Ok(#entries).*

The resulting set of proscribed dependencies for the purely optimistic and purely pessimistic approach can then be defined in terms of these lock classes as follows:

Optimistic Proscribed Dependencies:

**D1:** $T_i$: M(*str*) -->$_d$ $T_j$: M(*str*) $T_i$ modifies a key subsequently modified by $T_j$
**D2:** $T_i$: M(*str*) -->$_d$ $T_j$: L(*str*) $T_i$ modifies a key subsequently looked at by $T_j$
**D3:** $T_i$: M(*str*) -->$_d$ $T_j$: E $T_i$ modifies a key and $T_j$ subsequently "*Evals* ".

Pessimistic Proscribed Dependencies:

**D1:** $T_i$: M(*str*) -->$_d$ $T_j$: M(*str*) (same as above)
**D2:** $T_i$: M(*str*) -->$_d$ $T_j$: L(*str*) (same as above)

**D3:** $T_i$: M(*str*) -->$_d$ $T_j$: E (same as above)
**D4:** $T_i$: L(*str*) -->$_d$ $T_j$: M(*str*) (making the dependency **D2** symmetrical)
**D5:** $T_i$: E -->$_d$ $T_j$: M(*str*) (making the dependency **D3** symmetrical)

As with the *semiqueue* example, this type-specific treatment of conflicts thus defines a *directory* object as a collection of individually lockable elements.

We would expect the relative performance of optimistic and pessimistic treatment of this *directory* object to roughly correspond to the behavior exhibited by our implementation of optimistic and pessimistic *semiqueue* servers (section 6.1.1 and 6.1.2). That is, a purely optimistic technique can be expected to outperform a purely pessimistic technique at low levels of transaction conflict due to the inherent overhead associated with the increased number of pessimistic conflict types and its increased frequency of conflict checks (section 6.1.1). An exact threshold percentage of conflict where optimistic techniques begin to be outperformed by pessimistic techniques would be contingent on implementation dependent factors. Due to the similar nature of the operations associated with each object however, we would expect this to also roughly correspond to our *semiqueue* results.

The configuration of the three conflict types defined by **D1**, **D2** and **D3** for the optimistic server can be illustrated as follows:

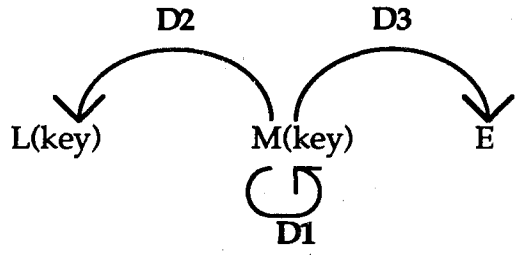



**Figure 6.2**: Directory Dependencies

Since there are three conflict types, there are $2^3$ possible optimistic/pessimistic conflict-type combinations (including purely optimistic and purely pessimistic schemes).

In situations where reliable information is available regarding the expected frequency of conflict types, a hybrid scheme could provide more effective concurrency control than either of the purely optimistic or pessimistic techniques. For example, when the data structure is seldom modified, but frequently looked up, **D2** should be treated pessimistically while **D1** and **D3** should be treated optimistically. This combination would lead to more efficient Eval and Modify operations than a purely pessimistic scheme, provided that the level of conflict for **D1** and **D3** remained low, and yet resolve conflicts of type **D2** more effectively than an purely optimistic scheme. In this example, the hybrid server could be expected to outperform both the optimistic and pessimistic schemes.

# 7 Conclusions and Further Research

## 7.1 Conclusions

Extended transactions have the potential to be a valuable tool for organizing and structuring computations in general purpose distributed systems [Herlihy86, Spector83, Spector85, Spector87, Weihl85]. The role of optimistic concurrency control in synchronization schemes that attempt to exploit type-specific properties of abstract objects within this vast application domain is largely undefined and unexplored. [Herlihy86] presents new type-specific optimistic techniques for objects in distributed systems that can be applied selectively in conjunction with pessimistic techniques on a per conflict-type basis. The results from our implementation of optimistic, pessimistic and hybrid *semiqueue* servers based on Herlihy's synchronization method confirm the conjecture that optimistic concurrency control may yet have a place in general purpose systems.

It is not unreasonable to assume that some reliable information regarding the expected frequency of conflict types would be available to the implementor of a shared abstract object. The customizable method of type-specific concurrency control studied here presents a means of exploiting this information and devising the most effective approach to concurrency control based on this knowledge. Our simulation served to demonstrate how this method can provide an entire spectrum of viable

and effective means of synchronizing extended transactions, ranging from purely pessimistic to purely optimistic techniques, within independent data servers. We were also able to determine the most appropriate environment for each of the optimistic, pessimistic and hybrid implementations.

Our results demonstrate the inherent robustness of type-specific pessimistic techniques, establish an impressive threshold percentage between the optimistic and pessimistic techniques, and identify important idiosyncrasies of the hybrid approach. Unlike the complete transaction systems of TABS and Argus however, our focus was only on one aspect of a general purpose distributed transaction facility: concurrency control within a single data server. Although we were able to establish the worthiness of optimistic methods within independent data servers, the impact the introduction of these techniques have on a general purpose distributed system as a whole is as yet undetermined.

The goal of research in this area is to simplify the chore of implementing correct and efficient synchronization properties for abstract objects used in distributed applications. A general purpose distributed transaction facility however, not only has to be simple to use, but it needs to provide efficient and flexible support for user defined abstract types. Implementors should be able to choose how they want their abstract objects to be treated. Towards this goal, the distributed transaction processing system, Camelot [Spector87], (the successor to the TABS prototype [Spector85]), supports two compatible types of synchronization: standard

pessimistic locking and hybrid atomicity, which has features of both timestamps and locking. Within single data servers, synchronization implementations can be tailored to provide pessimistic type-specific locking [Spector87]. We believe that the inclusion of the optimistic techniques studied here into general purpose systems such as this would provide users with a more complete spectrum of synchronization methods and consequently add to the efficiency and flexibility of the system as a whole .

## 7.2 Further Research

Possible directions for further research include the implementation of a protocol for validating distributed transactions, the development of an appropriate user interface for specifying type-specific consistency constraints, and the exploration of the ramifications associated with the use of pre-validation transaction abort in the optimistic approach.

To be compatible with pessimistic methods such as two phase locking which serialize transactions in commit order, the optimistic method studied here must not only validate transactions in commit timestamp order but also apply transactions' intentions lists in that order. Within our simulation model, the local*Validation* and subsequent *WRITE* phase of a committing transaction are implemented within

the same procedure, ensuring that only one transaction at a time can validate at a data server and that intentions lists are automatically applied in the the same order that transactions validated.

With distributed transactions however, a transaction could potentially require validation on several distinct sites before being able to procede to its *WRITE* phase. Consequently, validation of distributed transactions requires a much more rigorous protocol than our "automatic" approach. The expanded version of [Herlihy86][6] presents two possible protocols for validating distributed transactions, and addresses the issue of recoverability for the optimistic method studied here.

Since the purpose of a high performance extended transaction facility is to reduce the complexity associated with implementing shared abstract objects in distributed applications, a major priority is to make these facilities easy to use. The user interfaces of TABS and Camelot consist of system libraries to provide low level primitives which enable users to tailor their servers and explicitly control the setting of locks. Argus, on the other hand, hides some of its synchronization facilities in a language run-time system that mechanically imposes transparent support for built-in atomic types, and relies on the aid of a *mutex* object for the implementation of nontrivial servers. Although Argus is regarded as being easier to use for simple objects, it is difficult to compare the amount of work required by each approach to

---

[6] Received through personal correspondence.

implement more complex servers, and a formal performance comparison is not yet available [Spector85]. It is possible, however, that flexibility and efficiency have been sacrificed in Argus' higher level approach for the sake of user-friendliness. The greatest challenge associated with developing an appropriate user interface for specifying complex type-specific consistency constaints therefore lies with striking a balance between flexibility, efficiency and user-friendliness.

As previously discussed (section 4.2.1.1), the exploitation of state based information and pre-validation transaction abort could potentially reduce the amount of wasted work performed by the optimistic semiqueue server if the validating transaction must be aborted. Although this approach could be more accurately classified as a drastic pessimistic method that relies on transaction abort instead of delay, it could substantially improve our optimistic server's performance. The potential problem assoicated with this exploitation of optimistic locking information during the *READ* phase of a transaction is the overhead introduced by conflict detection. In its purest form, an optimistic approach offers a cost effective alternative to pessimistic methods under conditions where the level of conflict is sufficiently low by virtue of the fact that it contains less locking overhead. Since conflict detection contributes to locking overhead, allowing it in the *READ* phase may cancel out the advantages introduced by the optimistic method. Further exploration into these performance tradeoffs is required before the application of this technique could be considered to be generally applicable within type-specific

optimistic methods.

# Appendix

```
# ==========================
# The Pessimistic Semiqueue Server
# ==========================

resource Queue

  op Enq(Tnum:int; i:int) returns Enq_res:int          {call}
  op Eval(Tnum:int) returns item_count:int             {call}
  op Deq_rand(Tnum:int) returns rand:int               {call}
  op Deadlock_Check(blocker:int; Tnum:int)
    returns aborted:bool                               {call}
  op Abort(Tnum:int)                                   {call}
  op Global_Update(Tnum:int) returns glob:bool         {call}

  op Wakeup_Enq()              #Wakeup Calls
  op Wakeup_Eval()
  op Wakeup_DF()
  op Wakeup_DOK()
  op Wakeup_FOK()

  op Enq_awoke()               #Awoke repsonses
  op DF_awoke()
  op DOK_awoke()
  op Eval_awoke()
  op FOK_awoke()

body Queue()

  const N := 100               # N is the maximum number of concurrently
                               # transactions this server can support

  type node = rec(prev_link:ptr node                   # Nodes of the semiQ
            value:int
          Deq_Plock:int
          next_link:ptr node)

  type Deq_ptr_node = rec(Deq_link: ptr node           # Nodes of Deq intnts
                nxt: ptr Deq_ptr_node)

  var Qhead, Qtail, element: ptr node                  # Ptrs for accessing the semiQ
  var Deq_Failed_Plock[1:N]: bool                      # P-locks for Deq/Failed events
  var Eval_Plock[1:N]:    bool                         # P-locks for Eval/#items events
  var Enq_Plock[1:N]:     bool                         # P-locks for Enq/Ok events
  var head[1:N], tail[1:N]: ptr node                   # Ptrs for Enq intentions lists
  var Deq_Pointers[1:N]:   ptr Deq_ptr_node            # Ptrs for Deq intentions lists

  var retry_enq[1:N]:     bool                         # Flag for blocked Enq op
```

```
  var retry_eval[1:N]:     bool                    # Flag for blocked Eval op
  var retry_df[1:N]:       bool                    # Flag for blocked Deq/Failed
  var retry_dok[1:N]:      bool                    # Flag for blocked Deq/Ok
  var retry_fok[1:N]:      bool                    # Flag for blocked Deq op

initial
  Qhead := null                                    # Initialize SemiQ head pointer
  Qtail := null                                    # Initialize SemiQ tail pointer
  fa i:= 1 to N ->                                 # For all Tnums initialize:
    Deq_Failed_Plock[i] := false                   # Deq/Failed P-lock
    Eval_Plock[i] := false                         # Eval/(#items) P-lock
    Enq_Plock[i] := false                          # Enq(i)/Ok P-lock
    head[i] := null                                # Enq intentions head
    tail[i] := null                                # Enq intentions tail
    Deq_Pointers[i] := null                        # Deq intentions
    retry_eval[i] := false                         # Flags for blocked ops
    retry_enq[i] := false
    retry_df[i] := false
    retry_dok[i] := false
    retry_fok[i] := false
  af
end



#=====================================
#   Abort -- called by Deadlock_Check
#=====================================

proc Abort(Tnum)

var local_element: ptr node                        # Ref to SemiQ and Enq intentions
var Deq_temp: ptr Deq_ptr_node                     # Ref to Deq intentions
var dequed_items: bool
var DF_flag: bool                  # Flags DF Plock
var Enq_flag: bool                 # Flags Enq Plock
var Eval_flag: bool                # Flags Eval Plock

if head[Tnum] != null ->                           # If Enq intentions is not empty
  local_element:= head[Tnum]                       # Throw it away
  do local_element != null ->
    head[Tnum] := local_element^.next_link
    free(local_element)
    local_element:= head[Tnum]
  od
fi

dequed_items:= false
if Deq_Pointers[Tnum] != null ->                   # If Deq intentions is not empty
```

```
  Deq_temp:= Deq_Pointers[Tnum]                    # Throw it away
  do Deq_temp != null ->
    Deq_temp^.Deq_link^.Deq_Plock := 0
    Deq_Pointers[Tnum] := Deq_temp^.nxt
    free(Deq_temp)
    Deq_temp := Deq_Pointers[Tnum]
  od
  dequed_items:= true              # Flag the Deq
fi

retry_dok[Tnum] := false           # Re-init blocked flags
retry_df[Tnum]:= false
retry_enq[Tnum] := false
retry_eval[Tnum] := false
retry_fok[Tnum] := false

DF_flag:= false
if Deq_Failed_Plock[Tnum] ->                       # If Tnum has Deq/Failed lock
  Deq_Failed_Plock[Tnum] := false                  # Release it
  DF_flag:= true                                   # Flag DF Plock
fi

Enq_flag:= false
if Enq_Plock[Tnum] ->                              # If Tnum has Enq P-lock
  Enq_Plock[Tnum] := false                         # Release it
  Enq_flag:= true                                  # Flag Enq Plock
fi

Eval_flag:= false
if Eval_Plock[Tnum] ->                             # If Tnum had Eval P-lock
  Eval_Plock[Tnum] := false                        # Release it
  Eval_flag := true                                # Flag Eval Plock
fi

if DF_flag or Eval_flag ->                         # If DF or Eval Plock released
  fa i:=1 to N st i!=Tnum ->
    if retry_enq[i] -> Wakeup_Enq()                # Wake-up all blocked enq ops
                receive Enq_awoke()
    fi
  af
fi

if Enq_flag ->                                     # If Enq Plock released
  fa i:=1 to N st i!=Tnum ->
    if retry_df[i] -> Wakeup_DF()                  # Wake-up all blocked Deq/Fails
                receive DF_awoke()
    fi
  af
fi
if Enq_flag or dequed_items ->                     # If Enq Plock or shared elmts released
```

```
  fa i:=1 to N st i!=Tnum ->
    if retry_eval[i] -> Wakeup_Eval()              # Wake-up all blocked Eval ops
                receive Eval_awoke()
    fi
  af
fi

if dequed_items ->                                 # If Tnum had Deqed items
  fa i:= 1 to N st i!=Tnum ->
    if retry_fok[i] -> Wakeup_FOK()                # Wake-up all blocked Deq ops
                receive FOK_awoke()
    fi
  af
fi

if Eval_flag ->
  fa i:= 1 to N st i!= Tnum->
    if retry_dok[i] -> Wakeup_DOK()                # Wake-up all blocked Deq/Oks
                receive DOK_awoke()
    fi
  af
fi

end



# ============================================================
#   Deadlock_Checker -- invoked before any transaction blocks
# ============================================================

proc Deadlock_Check(blocker,Tnum) returns aborted

    # "blocker" is the Tnum of the transaction that holds a
    #  P-lock of the type "Tnum" is about to block on...

 aborted:= false
 if (retry_dok[blocker] or                         # If blocker is waiting in a Deq/Ok event
    retry_enq[blocker]) and                        # or blocker is waiting in an Enq/Ok event
    Eval_Plock[Tnum] ->                            # and Tnum holds an Eval P-lock
   write("ABORTING TRANSACTION",Tnum,"on Eval_Plock")
   Abort(Tnum)                                     # Abort Tnum
   aborted:= true
 [] retry_df[blocker] and                          # If blocker is waiting in a Deq/Failed event
   Enq_Plock[Tnum] ->                              # and Tnum holds a P-lock
   write("ABORTING TRANSACTION",Tnum,"on Enq_Plock")
   Abort(Tnum)                                     # Abort Tnum
   aborted:= true
 [] retry_eval[blocker] and          # If blocker is waiting in an Eval/#items
   (Enq_Plock[Tnum] or                             # and Tnum has an Enq P-lock or Deq P-lock
```

```
   Deq_Pointers[Tnum]!=null) ->
   write("ABORTING TRANSACTION",Tnum,"on Enq or Deq Plock")
   Abort(Tnum)                                    # Abort Tnum
   aborted:= true
 [] retry_enq[blocker] and          # If blocker is waiting in an Enq/Ok event
   Deq_Failed_Plock[Tnum] ->    # and Tnum has a Deq Failed P-lock
   write("ABORTING TRANSACTION",Tnum,"on Deq Failed Plock")
   Abort(Tnum)                                    # Abort Tnum
   aborted:= true
 [] retry_fok[blocker] and                        # If blocker is waiting in a Deq event
   Deq_Pointers[Tnum] != null ->                  # and Tnum has Deqed items
   write("ABORTING TRANSACTION",Tnum,"on Deq FOK")
   Abort(Tnum)                                    # Abort Tnum
   aborted:= true
 fi

end


# ======================
#     Enq(i)/Ok Event
# ======================

proc Enq(Tnum,i) returns Enq_res
  var local_element: ptr node
  var abort: bool

    retry_enq[Tnum]:= false        #  Initialize waiting flag

    fa i:= 1 to N st i!= Tnum ->
      if Deq_Failed_Plock[i] or    # If any trans. holds a Deq/Failed lock
         Eval_Plock[i] ->          # or an Eval P-lock
        retry_enq[Tnum] := true    # Set the waiting flag
        exit
      fi
    af

    do retry_enq[Tnum] ->                         # If there is a conflict...
      abort := false
      fa i:= 1 to N st i!= Tnum ->                # Deadlock chck before blocking
        if Deq_Failed_Plock[i] ->
          abort:= Deadlock_Check(i,Tnum)
          if abort -> exit fi
        fi
        if Eval_Plock[i] ->
          abort:= Deadlock_Check(i,Tnum)
          if abort -> exit fi
        fi
      af
```

```
      if abort -> Enq_res:= -9999                    # Resolve deadlock with abort
       return                          # of calling transaction
      fi

      Enq_res := -666                 # If no deadlock, WAIT
      reply
      receive Wakeup_Enq()                           # (for a wake-up call)
      retry_enq[Tnum]:= false
      fa i:= 1 to N st i!= Tnum ->                   # Unblocked -> conflict check
        if Deq_Failed_Plock[i] or
              Eval_Plock[i] ->
         retry_enq[Tnum] := true                     # If conflict, block again
         exit
        fi
      af
      send Enq_awoke()
    od
                                                     # When not blocked:
    Enq_Plock[Tnum]:= true                           # Assign an Enq P-lock
    local_element:= new(node)                        # Create a new node
    local_element^.value:=i                          # Assign it the value
    local_element^.Deq_Plock := 0                    # Init Deq P-lock for node
    local_element^.prev_link:= null                  # Init pointers for node
    local_element^.next_link:= null

    if head[Tnum] = null ->                          # Add node to Enq intnts list
      head[Tnum]:= local_element
      tail[Tnum]:= local_element

    [] else ->
      tail[Tnum]^.next_link:= local_element
      local_element^.prev_link:= tail[Tnum]
      tail[Tnum]:= local_element
    fi
    Enq_res:= 1                     # Return "ok" result
end

# ==========================
#  Eval/(#items) event
# ==========================

proc Eval(Tnum) returns item_count

  var local_element: ptr node                        # Ref to SemiQ and Enq intnts
  var local_deq: ptr Deq_ptr_node                    # Ref to Deq intentions
  var abort: bool

    retry_eval[Tnum] := false
    fa i:= 1 to N st i!=Tnum ->       # Check all other transactions
      if Deq_Pointers[i] != null or  # for conflicting locks
```

```
      Enq_Plock[i] ->
    retry_eval[Tnum] := true
    exit
  fi
 af

 do retry_eval[Tnum] ->          # If a conflict exists
   abort := false
   fa i:= 1 to N st i!= Tnum ->  # Do a Deadlock check
     if Deq_Pointers[i]!=null ->
       abort:= Deadlock_Check(i,Tnum)
       if abort -> exit fi                    # Abort if necessary
     fi
     if Enq_Plock[i] ->
       abort:= Deadlock_Check(i,Tnum)
       if abort -> exit fi
     fi
   af

   if abort ->                                # If aborted, return -9999
     item_count:= -9999
    return
   fi

   item_count :=-666
   reply
   receive Wakeup_Eval()       # Wait for wakeup call
   retry_eval[Tnum]:= false
   fa i:= 1 to N st i!= Tnum ->  # When unblocked, check all trans
     if Deq_Pointers[i] != null  # for conflicts
         or Enq_Plock[i] ->
       retry_eval[Tnum] := true             # If conflicts -> block again
       exit
     fi
   af
    send Eval_awoke()
 od

 Eval_Plock[Tnum]:= true       # When unblocked, assign Eval P-lock
 item_count:=0
 element:= Qhead

 do (element != null) ->       # Count items in semiQ
     item_count++
     element:= element^.next_link
 od

 local_element:= head[Tnum]                 # Add elements from Enq intentions
 do (local_element != null) ->
     item_count++
```

```
        local_element:= local_element^.next_link
    od

    local_deq := Deq_Pointers[Tnum]                    # Subtract elements in Deq intnts
    do (local_deq != null) ->
       item_count--
       local_deq := local_deq^.nxt
    od

end


# ====================================
#     Deq/Ok(i) and Deq/Failed Events
# ====================================

proc Deq_rand(Tnum) returns rand

    var abort:  bool
    var start_again : bool
    var found_one : bool
    var num_dequed : int
    var num_queued : int
    var local_element: ptr node
    var Deq_temp: ptr Deq_ptr_node

# It is ok to play in your own back yard.

    if head[Tnum] != null ->                           # If Enq intentions not empty
       local_element:= head[Tnum]                      # Deq an element from there
       rand:= local_element^.value
       head[Tnum] := local_element^.next_link
       if head[Tnum] = null ->                         # If 0 elements now in Enq int.
          Enq_Plock[Tnum]:=false                       # Release P-lock
          fa i:=1 to N st i!= Tnum ->
             if retry_df[i] -> Wakeup_DF()             # Wake-up blocked Deq/Failed
                    receive DF_awoke()
          fi
       af
       fi
       free(local_element)
       return

    [] else ->
       start_again := true          # Flag to restart Deq op
       retry_df[Tnum] := true                          # Flag to restart Deq/Failed

       do start_again ->

          do retry_df[Tnum] ->
```

```
 num_dequed:= 0
 num_queued:= 0
  Deq_temp:= Deq_Pointers[Tnum]
  do Deq_temp != null ->                    # Count items in Deq intnts
   num_dequed++
    Deq_temp := Deq_temp^.nxt
 od
  element := Qhead
  do element != null ->       # Count items in semiQ
   num_queued++
    element := element^.next_link
 od

 if num_dequed = num_queued ->              # If no eligible items
   retry_df[Tnum] := false
   fa i:= 1 to N st i!= Tnum ->             # Check for conflicts
    if Enq_Plock[i] ->       # If some Enq P-lock exists
      retry_df[Tnum] := true                # Block Deq/Failed event
     exit
    fi
  af
   if retry_df[Tnum] = false ->             # If not blocked
    rand := -99
     Deq_Failed_Plock[Tnum] := true  # Deq/Failed P-locked
   return
  [] else ->                    # Otherwise, if blocked
    abort := false
    fa i:= 1 to N st i!= Tnum ->            # Do Deadlock check
     if Enq_Plock[i] ->
       abort:= Deadlock_Check(i,Tnum)
      if abort -> exit fi                   # If necessary, abort
     fi
    af

     if abort -> rand:= -9999; return fi
    rand :=-666
    reply
     receive Wakeup_DF()                    # Wait for Wakeup call
     send DF_awoke()
  fi

 [] else ->                     # If there are eligible elmnts:
   retry_df[Tnum] := false
 fi
od

retry_fok[Tnum] := true
do retry_fok[Tnum] ->
 found_one := false
 element := Qhead
```

```
 do element != null and ~found_one ->   # Search SemiQ
  if element^.Deq_Plock = 0 ->                # If unlocked element
   found_one := true                          # Flag it "found"...
  [] else ->
     element := element^.next_link
  fi
od

if found_one ->                               # If "found" elmnt
   retry_dok[Tnum] := false
   fa i:= 1 to N st i!=Tnum ->                # Check for conflict
      if Eval_Plock[i] ->
       retry_dok[Tnum]:=true
      exit
    fi
  af

  if retry_dok[Tnum] ->                       # If conflict exists,
     abort := false
     fa i:= 1 to N st i!= Tnum ->             # Do Deadlock check
        if Eval_Plock[i] ->
          abort:= Deadlock_Check(i,Tnum)
         if abort -> exit fi                  # Abort if necessary
      fi
    af

     if abort -> rand := -9999; return fi
    rand:= -666
    reply
     receive Wakeup_DOK()                     # Wait for Wakeup call
     send DOK_awoke()
  [] else ->                  # Otherwise, take item
     rand:= element^.value
     element^.Deq_Plock := 1                  # P-lock semiQ node
      Deq_temp:= new(Deq_ptr_node)     # Add to Deq intnts
      Deq_temp^.Deq_link:= element
     Deq_temp^.nxt:= Deq_Pointers[Tnum]
     Deq_Pointers[Tnum]:= Deq_temp
     retry_fok[Tnum] := false
   return
  fi

[] else ->                    # If elig items P-locked
   abort := false
   fa i:= 1 to N st i!= Tnum ->
     if Deq_Pointers[i] != null ->            # Do Deadlock check
       abort := Deadlock_Check(i,Tnum)
      if abort -> exit fi                     # Abort if necessary
   fi
  af
```

```
            if abort -> rand := -9999; return fi
            rand := -666
            reply
            receive Wakeup_FOK()                    # Wait for wakeup call
            send FOK_awoke()
            if Qhead = null ->                      # If semiQ is empty
              retry_df[Tnum] := true                # Retry Deq/Faild event
               retry_fok[Tnum] := false
            fi
          fi
        od
      od
    fi
end


#=====================================
#     Global_Update Proc
#=====================================

proc Global_Update(Tnum) returns glob
 var Deq_temp: ptr Deq_ptr_node
 var dequed_items: bool
 var Enq_flag:bool
 var DF_flag:bool
 var Eval_flag:bool

   if head[Tnum] != null ->                         # If Enq intentions not empty
     if Qhead != null ->            # Add it to SemiQ
        Qtail^.next_link := head[Tnum]
        head[Tnum]^.prev_link := Qtail
        Qtail := tail[Tnum]
     [] else ->
       Qhead := head[Tnum]
        Qtail := tail[Tnum]
     fi
     head[Tnum]:= null              # Re-init Enq intentions ptrs
     tail[Tnum]:= null
   fi

   dequed_items:= false
   if Deq_Pointers[Tnum] != null ->                 # If Deq intentions not empty
     dequed_items:= true
     Deq_temp := Deq_Pointers[Tnum]
     do Deq_temp != null ->                         # Remove referenced elements
       if Deq_temp^.Deq_link = Qhead ->
          Qhead:= Deq_temp^.Deq_link^.next_link
          if Qhead != null -> Qhead^.prev_link:= null fi
```

```
    [] Deq_temp^.Deq_link = Qtail and Qtail != Qhead ->
        Qtail:= Deq_temp^.Deq_link^.prev_link
        Qtail^.next_link:= null
    [] else ->
        Deq_temp^.Deq_link^.prev_link^.next_link :=
            Deq_temp^.Deq_link^.next_link
        Deq_temp^.Deq_link^.next_link^.prev_link :=
            Deq_temp^.Deq_link^.prev_link
    fi
    free(Deq_temp^.Deq_link)
    Deq_Pointers[Tnum] := Deq_temp^.nxt
    free(Deq_temp)
    Deq_temp := Deq_Pointers[Tnum]
  od
 fi

 Enq_flag := false
 if Enq_Plock[Tnum] ->                          # If Tnum had an Enq P-lock
   Enq_Plock[Tnum] := false                     # Release it
   Enq_flag:= true              # Flag Enq Plock
 fi

 DF_flag:= false
 if Deq_Failed_Plock[Tnum]  ->                  # If Tnum had Deq/Failed P-lock
    Deq_Failed_Plock[Tnum] := false             # Release it
   DF_flag:= true               # Flag Deq/Failed P-lock
 fi

 Eval_flag:= false
 if Eval_Plock[Tnum] ->                         # If Tnum had Eval P-lock
   Eval_Plock[Tnum] := false                    # Release it
   Eval_flag:= true             # Flag Eval P-lock
 fi

 if Enq_flag ->
   fa i:=1 to N st i!= Tnum ->                  # Issue wakeup calls
     if retry_df[i] -> Wakeup_DF()
              receive DF_awoke()
     fi
   af
 fi

 if DF_flag or Eval_flag ->
   fa i:=1 to N st i != Tnum ->                 # Issue wakeup calls
     if retry_enq[i] -> call Wakeup_Enq()
              receive Enq_awoke()
     fi
   af
 fi
```

```
    if Eval_flag ->
      fa i:=1 to N st i != Tnum ->                    # Issue wakeup calls
        if retry_dok[i] -> Wakeup_DOK()
                   receive DOK_awoke()
        fi
      af
    fi

    if dequed_items or Enq_flag ->                    # If Tnum had Deq/Ok(i) P-lock
      fa i:= 1 to N st i!=Tnum ->                     # Issue wakeup calls
        if retry_eval[i] ->  Wakeup_Eval()
                   receive Eval_awoke()
        fi
        if retry_fok[i] -> Wakeup_FOK()
                   receive FOK_awoke()
        fi
      af
    fi
  glob := true
  end
end
```

```
# ================================
# The Optimistic Semiqueue Server
# ================================


resource Queue

  op Enq(Tnum:int; i:int) returns Enq_res:int        {call}
  op Eval(Tnum:int) returns item_count:int           {call}
  op Deq_rand(Tnum:int) returns rand:int             {call}
  op Validate(Tnum:int)                              {call}
  op Abort(Tnum:int)                                 {call}

body Queue()

  const N := 100                                     # N is the maximum number of concurrently
                                                     # active transactions this server can support

  type node = rec(prev_link:ptr node                 # Nodes of the semiqueue
            value:int
          Deq_Olock[1:N+1]:int   # (treated as boolean)
          next_link:ptr node)

  type Deq_ptr_node = rec(Deq_link: ptr node  # Nodes of the Deq
                nxt: ptr Deq_ptr_node)               # intentions list

  var Qhead, Qtail, element: ptr node                # Ptrs for accessing the semiQ
  var Deq_Failed_Olock[1:N]:bool                     # O-locks for Deq/Failed events
  var Eval_Olock[1:N]:bool        # O-locks for Eval/(#items) events
  var head[1:N], tail[1:N]: ptr node                 # Ptrs for Enq intentions lists
  var Deq_Pointers[1:N]: ptr Deq_ptr_node            # Ptrs for Deq intentions lists


initial
  Qhead := null                                      # Initialize SemiQ head pointer
  Qtail := null                                      # Initialize SemiQ tail pointer
  fa i:= 1 to N ->                                   # For all Tnums initialize:
    Deq_Failed_Olock[i] := false                     # Deq/Failed O-locks
    Eval_Olock[i] := false                           # Eval/(#items) O-locks
    head[i] := null                  # Enq intentions head
    tail[i] := null                  # Enq intentions tail
    Deq_Pointers[i] := null                          # Deq intentions list
  af
end
```

```
#=======================
#     Enq(i)/Ok Event
#=======================

proc Enq(Tnum,i) returns Enq_res

  var local_element:ptr node

    local_element:= new(node)                    # Create a new node
    local_element^.value:=i                      # Assign it the calling value
    fa j:= 1 to N+1 ->
      local_element^.Deq_Olock[j] := 0           # Initialize node's O-locks
    af
    local_element^.prev_link:= null              # Initialize node's pointers
    local_element^.next_link:= null

    if head[Tnum] = null ->                      # If Enq intentions is empty
      head[Tnum]:= local_element                 # Set Enq intentions head
      tail[Tnum]:= local_element                 # Set Enq intentions tail

    [] else ->                       # Otherwise,
      tail[Tnum]^.next_link:= local_element  # append to Enq intnts
      local_element^.prev_link:= tail[Tnum]
      tail[Tnum]:= local_element
    fi

   Enq_res:= 1                       # Respond "ok" to calling Tnum
end



#=========================
#     Eval/(#items) Event
#=========================

proc Eval(Tnum) returns item_count

  var local_element: ptr node                    # Ref to semiQ and Enq intnts
  var local_deq: ptr Deq_ptr_node                # Ref to Deq intentions list

   Eval_Olock[Tnum]:= true                       # Set Eval O-lock for Tnum
   item_count:=0                   # Initialize item counter

    element:= Qhead                # Count elements in semiQ
    do (element != null) ->
      item_count++
      element:= element^.next_link
    od

    local_element:= head[Tnum]                   # Add elements in Enq intnts
```

```
      do (local_element != null) ->
        item_count++
        local_element:= local_element^.next_link
      od

      local_deq := Deq_Pointers[Tnum]          # Subtract Deq intentions list
      do (local_deq != null) ->
        item_count--
        local_deq := local_deq^.nxt
      od
end



#=====================================
#     Deq/Ok(i) and Deq/Failed Events
#=====================================

proc Deq_rand(Tnum) returns rand
   var num_items:int              # The number of eligible elements
   var local_element: ptr node                    # Ref to semiQ and Enq intentions
   var found_one: bool
   var Deq_temp: ptr Deq_ptr_node                 # Ref to Deq intentions

   if head[Tnum] != null ->                       # If Enq intntns is not empty
       local_element:= head[Tnum]                 # Remove head of Enq intnts
       rand:= local_element^.value
       head[Tnum] := local_element^.next_link
       free(local_element)
      return

   [] else ->                     # Otherwise,
       element := Qhead           # Start searching the SemiQ
       num_items:= 0
       found_one := false

       do (element != null) ->
         if element^.Deq_Olock[Tnum] =0 ->        # If not in Deq intnts
          num_items++                             # add to no. eligible
           if element^.Deq_Olock[N+1] = 0 ->      # If not already locked
            found_one := true                     # THIS IS IT
            exit                                  # So quit searching
         fi
        fi
         element:= element^.next_link
      od

      if num_items = 0 ->           # If no eligible elements,
         rand:= -99                               # rand = "failed"
          Deq_Failed_Olock[Tnum] := true          # Set Deq/Failed Olock
```

```
        return                                  # Return response

      [] ~found_one ->                          # If eligible el.s are locked
         element:=Qhead                         # Take the first one
         do element^.Deq_Olock[Tnum] = 1 ->
           element:= element^.next_link
         od
      fi

      rand:= element^.value                     # Assign value to rand
      element^.Deq_Olock[Tnum] := 1             # Set Tnum's Deq O-lock
      element^.Deq_Olock[N+1] := 1              # Set O-locked flag
      Deq_temp:= new(Deq_ptr_node)              # Add to Deq intentions
      Deq_temp^.Deq_link:= element
      Deq_temp^.nxt:= Deq_Pointers[Tnum]
      Deq_Pointers[Tnum]:= Deq_temp
  fi
end


#=============================================================
#     Abort(Tnum) -- invoked by Validate to resolve conflicts
#=============================================================

proc Abort(Tnum)

    var local_element: ptr node                 # Ref to SemiQ and Enq intnts
    var Deq_temp: ptr Deq_ptr_node              # Ref to Deq intentions
    var olocked: bool

    if head[Tnum] != null ->                    # If Enq intnts is not empty
      local_element:= head[Tnum]                # Throw it away
      do local_element != null ->
        head[Tnum] := local_element^.next_link
        free(local_element)
        local_element := head[Tnum]
     od
    fi

    if Deq_Pointers[Tnum] != null ->            # If Deq intnts is not empty
      Deq_temp:= Deq_Pointers[Tnum]             # Throw it away
      do Deq_temp != null ->
        Deq_temp^.Deq_link^.Deq_Olock[Tnum]:= 0   # Reset Deq O-lock
        olocked := false
        fa i:= 1 to N ->                    # Check O-locked flag
          if Deq_temp^.Deq_link^.Deq_Olock[i] = 1 ->
            olocked:= true
          fi
```

```
      af
      if ~olocked ->                              # Reset flag if nec.
         Deq_temp^.Deq_link^.Deq_Olock[N+1] := 0
      fi
      Deq_Pointers[Tnum] := Deq_temp^.nxt
      free(Deq_temp)
      Deq_temp := Deq_Pointers[Tnum]
    od
   fi

   Deq_Failed_Olock[Tnum] := false               # Re-init O-locks
   Eval_Olock[Tnum] := false

end



#==========================
#     Validation
#==========================

proc Validate(Tnum)

 var Enq_Items, Deq_Items: bool
 var Deq_temp: ptr Deq_ptr_node


   Enq_Items := false
   Deq_Items := false
   if head[Tnum] != null ->                    # If Enq intnts is not empty
     Enq_Items := true
     fa i:= 1 to N st i != Tnum ->             # Check all other Tnums
       if Deq_Failed_Olock[i]=true or          # for Deq/Failed
        Eval_Olock[i]=true ->                  # or Eval/(#items) O-locks
       Abort(Tnum)                             # Abort if there's conflict
       return
      fi
     af
   fi

   if Deq_Pointers[Tnum] != null ->            # If Deq intnts is not empty
     Deq_Items := true
    if ~Enq_Items ->                           # (If not already done --
      fa i:= 1 to N st i!=Tnum ->              #  check Eval O-locks and
        if Eval_Olock[i] ->
        Abort(Tnum)                            #  Abort if there's conflict)
        return
       fi
      af
    fi
```

```
 Deq_temp:= Deq_Pointers[Tnum]
 do Deq_temp != null ->                          # For each node to be Deqed
   fa i:=1 to N st i != Tnum ->                  # Check for Deq conflict
     if Deq_temp^.Deq_link^.Deq_Olock[i] = 1 ->
      Abort(Tnum)                                # Abort if necessary
      return
     fi
   af
   Deq_temp := Deq_temp^.nxt
 od
fi

if Enq_Items = true ->                           # Append Enq intentions
  if Qhead != null ->
    Qtail^.next_link := head[Tnum]
    head[Tnum]^.prev_link := Qtail
    Qtail := tail[Tnum]

  [] else ->
    Qhead := head[Tnum]
    Qtail := tail[Tnum]
  fi

 head[Tnum]:= null                               # Re-init Enq intentions
 tail[Tnum]:= null

fi

if Deq_Items = true ->                           # Remove refs in Deq intnts
  Deq_temp := Deq_Pointers[Tnum]
  do Deq_temp != null ->
    if Deq_temp^.Deq_link = Qhead ->
       Qhead:= Deq_temp^.Deq_link^.next_link
       if Qhead != null -> Qhead^.prev_link:= null fi
    [] Deq_temp^.Deq_link = Qtail and Qtail != Qhead ->
       Qtail:= Deq_temp^.Deq_link^.prev_link
       Qtail^.next_link:= null
    [] else ->
       Deq_temp^.Deq_link^.prev_link^.next_link :=
            Deq_temp^.Deq_link^.next_link
       Deq_temp^.Deq_link^.next_link^.prev_link :=
            Deq_temp^.Deq_link^.prev_link
   fi
    free(Deq_temp^.Deq_link)
    Deq_Pointers[Tnum] := Deq_temp^.nxt
    free(Deq_temp)
    Deq_temp := Deq_Pointers[Tnum]
 od

 fi
```

```
    Deq_Failed_Olock[Tnum] := false          # Re-init O-locks
    Eval_Olock[Tnum] := false
end
end
```

```
#================================
#     The Hybrid Server
#================================


resource Queue

  op Enq(Tnum:int; i:int) returns Enq_res:int          {call}
  op Eval(Tnum:int) returns item_count:int             {call}
  op Deq_rand(Tnum:int) returns rand:int               {call}
  op Validate(Tnum:int) returns validated:bool         {call}
  op Deadlock_Check(blocker:int; Tnum:int)
    returns aborted:bool                               {call}
  op Abort(Tnum:int)                                   {call}
  op Global_Update(Tnum:int) returns glob:bool         {call}

  op Wakeup_FOK()

  op FOK_awoke()

body Queue()

  const N := 100                  # N is the maximum number of concurrently active
                                  # transactions this server can support

  type node = rec(prev_link:ptr node                   # Nodes of the semiqueue
            value:int
          Deq_Plock:int
          next_link:ptr node)

  type Deq_ptr_node = rec(Deq_link: ptr node  # Nodes of Deq intnts
                nxt: ptr Deq_ptr_node)

  var Qhead, Qtail, element: ptr node                  # Ptrs for accessing semiQ
  var Deq_Failed_Olock[1:N]: bool                      # O-lock for Deq/Failed event
  var Eval_Olock[1:N]:     bool                        # O-lock for Eval/#items event
  var head[1:N], tail[1:N]:  ptr node                  # Ptrs for Enq intentions lists
  var Deq_Pointers[1:N]:    ptr Deq_ptr_node           # Ptrs for Deq intnts lists

  var retry_fok[1:N]:      bool                        # Flag for blocked Deq op

initial
  Qhead := null                                        # Initialize SemiQ head pointer
  Qtail := null                                        # Initialize SemiQ tail pointer
  fa i:= 1 to N ->                                     # For all Tnums:
    Deq_Failed_Olock[i] := false                       # Init Deq/Failed O-lock
    Eval_Olock[i] := false                             # Init Eval/#items O-lock
    head[i] := null                                    # Init Enq intentions list head
    tail[i] := null                                    # Init Enq intentions list tail
    Deq_Pointers[i] := null                            # Init Deq intentions list
```

```
    retry_fok[i] := false                          # Init Deq blocked flag
  af
end

#=====================================================
#     Abort -- called by Validate and Deadlock_Check
#=====================================================

proc Abort(Tnum)

var local_element: ptr node                        # Ref to SemiQ and Enq intentions nodes
var Deq_temp: ptr Deq_ptr_node                     # Ref to Deq intentions
var dequed_items: bool

if head[Tnum] != null ->            # If Enq intentions list is not empty
  local_element:= head[Tnum]                       # Throw it out
  do local_element != null ->
    head[Tnum] := local_element^.next_link
    free(local_element)
    local_element:= head[Tnum]
  od
fi

dequed_items:=false
if Deq_Pointers[Tnum] != null ->                   # If Deq intentions list is not empty
  Deq_temp:= Deq_Pointers[Tnum]
  do Deq_temp != null ->                           # Throw it out
    Deq_temp^.Deq_link^.Deq_Plock := 0             # Release P-locks
    Deq_Pointers[Tnum] := Deq_temp^.nxt
    free(Deq_temp)
    Deq_temp := Deq_Pointers[Tnum]
  od
  dequed_items:=true                # Flag Deq attempt
fi

retry_fok[Tnum] := false

if dequed_items ->                  # If Deq had been attempted
  fa i:=1 to N st i!=Tnum ->        # Wakeup all blocked Deq ops
    if retry_fok[i] -> Wakeup_FOK()
                receive FOK_awoke()
    fi
  af
fi

Eval_Olock[Tnum] := false           # Release all O-locks
Deq_Failed_Olock[Tnum] := false
end
```

```
#================================================================
#     Deadlock_Check – invoked before any transaction blocks
#================================================================

proc Deadlock_Check(blocker,Tnum) returns aborted

    # "blocker" is the trans. number of the transaction "Tnum" is
    # about to block on.
 aborted:= false
 if retry_fok[blocker] and          # If "blocker" is waiting on Deq P-lock
   Deq_Pointers[Tnum] != null ->                    # and Tnum holds one
   Abort(Tnum)                      # then abort
   aborted:= true

 fi

end


#=======================================
#     Enq/Ok(i) Event
#=======================================

proc Enq(Tnum,i) returns Enq_res

  var local_element:ptr node                    # Reference to Enq intentions node

    local_element:= new(node)                   # Create new Enq intentions node
    local_element^.value:=i                     # Assign it a value
    local_element^.Deq_Plock := 0               # Init P-lock
    local_element^.prev_link:= null             # Init pointers
    local_element^.next_link:= null

    if head[Tnum] = null ->                     # Add node to Enq intentions list
      head[Tnum]:= local_element
      tail[Tnum]:= local_element

    [] else ->
      tail[Tnum]^.next_link:= local_element
      local_element^.prev_link:= tail[Tnum]
      tail[Tnum]:= local_element
    fi
    Enq_res:= 1                     #Return Enq "Ok"
end


#================================
#     Eval/(#items) Event
#================================
```

```
proc Eval(Tnum) returns item_count

  var local_element: ptr node      # Pointer to SemiQ nodes and Enq nodes
  var local_deq: ptr Deq_ptr_node                  # Pointer to Deq intentions

   Eval_Olock[Tnum]:= true         # Assign an O-lock for Eval/no. event
   item_count:=0                   # Init item counter
   element:= Qhead                 # Init ptr to SemiQ head

   do (element != null) ->         # Count number of items in SemiQ
      item_count++
       element:= element^.next_link
   od

   local_element:= head[Tnum]                  # Add number of items in Enq intnts
   do (local_element != null) ->
      item_count++
       local_element:= local_element^.next_link
   od

   local_deq := Deq_Pointers[Tnum]             # Subtract no. of items in Deq intnts
   do (local_deq != null) ->
      item_count--
       local_deq := local_deq^.nxt
   od

end


#=========================================
#     Deq/Failed and Deq/Ok(i) Events
#=========================================

proc Deq_rand(Tnum) returns rand

   var local_element: ptr node                 # Ptr to Enq intentions list
   var Deq_temp: ptr Deq_ptr_node              # Ptr to Deq intentions list
   var abort: bool                             # Flag to abort
   var start_again: bool                       # Flag to restart after block
   var found_one: bool                         # Flag an element to Deq
   var num_dequed: int                         # Total number in Deq intnts
   var num_queued: int                         # Total number in SemiQ

# It is ok to play in your own back yard.

   if head[Tnum] != null ->                    # If Enq intentions not empty
      local_element:= head[Tnum]               # Take an element from there
       rand:= local_element^.value
       head[Tnum] := local_element^.next_link
```

```
    free(local_element)
   return

[] else ->                          # Otherwise, check SemiQ
    start_again:= true

    do start_again ->

     num_dequed := 0                          # Init Deqed count
     num_queued := 0                          # Init Enqed count
      Deq_temp := Deq_Pointers[Tnum]
      do Deq_temp != null ->
       num_dequed++                           # Total Dequed
        Deq_temp := Deq_temp^.nxt
     od
      element := Qhead
      do element != null ->
       num_queued++                           # Total in SemiQ
        element := element^.next_link
     od
      if num_queued = num_dequed ->           # If Tnum has Deqed all
        Deq_Failed_Olock[Tnum] := true        # O-lock Deq/Fail
       rand := -99
       return
      fi

    retry_fok[Tnum] := true
    do retry_fok[Tnum] ->                     # If there's an item Tnum hasn't Deqed
     found_one := false
     element := Qhead
     do element != null and ~found_one ->     # Find an unlocked one
       if element^.Deq_Plock = 0 ->
        found_one := true
       [] else -> element := element^.next_link
       fi
     od

     if found_one ->              # If there is an unlocked one
       rand:= element^.value                  # Take it!
       element^.Deq_Plock := 1
       Deq_temp:= new(Deq_ptr_node)
       Deq_temp^.Deq_link:= element
       Deq_temp^.nxt:= Deq_Pointers[Tnum]
       Deq_Pointers[Tnum]:= Deq_temp
        # write("Transaction",Tnum,"Dequed item:",rand)
       retry_fok[Tnum]:= false
      return

     [] else ->                   # If there isn't one unlocked
       abort := false
```

```
        fa i:= 1 to N st i!=Tnum ->                    # Do a deadlock check
          if Deq_Pointers[i] != null ->
            abort := Deadlock_Check(i,Tnum)
            if abort -> exit fi
        fi
       af
        if abort -> rand := -9999; return fi
       rand:= -666
       reply
        receive Wakeup_FOK()                           # Wait if necessary
        send FOK_awoke()
        if Qhead = null ->
          retry_fok[Tnum] := false
       fi
      fi
     od
    od
  fi
end


#=======================================
#     Validation
#=======================================


proc Validate(Tnum) returns validated

 var Deq_temp: ptr Deq_ptr_node                        # Ref to Deq intentions

 validated := true
 if head[Tnum] != null ->                              # If Enq intentions not empty
   fa i:=1 to N st i!=Tnum ->                          # Check for conflicts
     if Eval_Olock[i]=true or Deq_Failed_Olock[i]=true ->
      Abort(Tnum)                  # Abort if conflict exists
       validated:= false
      return
     fi
   af
 fi

 if Deq_Pointers[Tnum] != null ->                      # If Deq intentions not empty
   Deq_temp:= Deq_Pointers[Tnum]
   do Deq_temp != null ->
     fa i:= 1 to N st i!=Tnum ->                       # Check for conflicts
       if Eval_Olock[i] = true ->
        Abort(Tnum)                  # Abort if there is one
         validated := false
        return
       fi
```

```
    af
    Deq_temp := Deq_temp^.nxt
  od
 fi
end



#=======================================
#     Global_Update
#=======================================

proc Global_Update(Tnum) returns glob
 var Deq_temp: ptr Deq_ptr_node
 var dequed_items: bool


  if head[Tnum] != null ->                          # If Enq intentions not empty
    if Qhead != null ->              # Append Enq intentions->SemiQ
      Qtail^.next_link := head[Tnum]
      head[Tnum]^.prev_link := Qtail
      Qtail := tail[Tnum]

    [] else ->
      Qhead := head[Tnum]
      Qtail := tail[Tnum]
    fi

    head[Tnum]:= null                # Re-init Enq intnts ptrs
    tail[Tnum]:= null
  fi
  dequed_items:= false
  if Deq_Pointers[Tnum] != null ->                  # If Deq intentions not null
    dequed_items:= true
    Deq_temp := Deq_Pointers[Tnum]
    do Deq_temp != null ->                          # Remove items from SemiQ
      if Deq_temp^.Deq_link = Qhead ->
        Qhead:= Deq_temp^.Deq_link^.next_link
        if Qhead != null -> Qhead^.prev_link:= null fi
      [] Deq_temp^.Deq_link = Qtail and Qtail != Qhead ->
        Qtail:= Deq_temp^.Deq_link^.prev_link
        Qtail^.next_link:= null
      [] else ->
        Deq_temp^.Deq_link^.prev_link^.next_link :=
            Deq_temp^.Deq_link^.next_link
        Deq_temp^.Deq_link^.next_link^.prev_link :=
            Deq_temp^.Deq_link^.prev_link
      fi
      free(Deq_temp^.Deq_link)
      Deq_Pointers[Tnum] := Deq_temp^.nxt
```

```
      free(Deq_temp)
      Deq_temp := Deq_Pointers[Tnum]
    od

    fi

    if dequed_items ->              # If Tnum Deqed items
      fa i:=1 to N st i != Tnum ->                    # Wake up any blocked Deq ops
        if retry_fok[i] -> call Wakeup_FOK()
                  receive FOK_awoke()
      fi
     af
    fi

    Eval_Olock[Tnum] := false                         # Release all O-locks
     Deq_Failed_Olock[Tnum] := false
    glob := true

 end
end
```

# References

[Agrawal87] Rakesh Agrawal, Michael Cary and Miron Livny, "Concurrency Control Performance Modeling: Alternatives and Implications", in *ACM Transactions on Database Systems* , 12(4), December 1987.

[Andrews87] Gregory R. Andrews and Ronald A. Olsson, "Revised Report on the SR Programming Language", University of Arizona TR 87-27, 1987.

[Andrews88] Gregory R. Andrews et al., "An Overview of the SR Language and Implmentation", in *ACM Transactions on Programming Languages and Systems*, 10(1), January 1988.

[Cordon85] Richardo Cordon and Hector Garcia-Molina, "The Performance of a Concurrency Control Mechanism that Exploits Semantic Knowledge", in *Proceedings of the Fifth International Conference on Distributed Computing Systems*, 1985.

[Eswaran76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System", *Communications ACM* , 19(11), November 1976.

[Franaszek85] P. Franaszek and J.T. Robinson, "Limitations of Concurrency in Transaction Processing", in *ACM Transactions on Database Systems*, 10(1), March 1985.

[Gawlick85] Dieter Gawlick, "Processing Hot Spots in High Performance Systems", In *Proceedings Compcon85*, 1985.

[Harder83] T. Harder,"Observations on Optimistic Concurrency Control Schemes", in *Information Systems*, 9, June 1984.

[Herlihy86] Maurice Herlihy, "Optimistic Concurrency Control for Abstract Data Types", in *Proceedings of the Principles of Distributed Computing Conference* , 1986.

[Kung81] H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control", in *ACM Transactions on Database Systems*, 6(2), June 1981.

[Liskov87] Barbara Liskov et al., "Argus Reference Manual", Massachusetts Institute of Technology TR-400, November 1987.

[Liskov83] Barbara Liskov and Robert Scheifler, "Gaurdians and Actions: Linguistic Support for Robust, Distributed Programs", in *ACM Transactions on Programming Languages and Systems* , 5(3), July 1983.

[O'Neil86] Patrick O'Neil, "The Escrow Transactional Method", in *ACM Transactions on Database Systems*, 11(4), December 1986.

[Pun87] K.H. Pun and G.G. Belford, "Performance Study of Two Phase Locking in Single-Site Database Systems", in *IEEE Transactions on Software Engineering* , 13(12), December 1987.

[Schwarz84] Peter M. Schwarz and Alfred Z. Spector, "Synchronizing Shared Abstract Types", in *ACM Transactions on Computer Systems*, 2(3), August 1984.

[Spector83] Alfred Z. Spector and Peter M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing", in *ACM Operating Systems Review*, 17(2), April 1983.

[Spector85] Alfred Z. Spector et al., "Distributed Transactions for Reliable Systems", in *Proceedings of the Principles of Distributed Computing Conference*, 1985.

[Spector87] Alfred Z. Spector, "Distributed Processing and the Camelot System", Carnegie-Mellon University TR-87-100, Jaunuary 1987.

[Thanos83] C. Thanos, C. Carlesi and E. Bertino, "Performance Evaluation of Two-Phase Locking Algorithms in a System for Distributed Databases", in *The Third Symposium on Reliability in Distributed Systems and Database Systems*, October 1983.

[Vidyasankar84] K. Vidyasankar and V. Raghavan, "Highly Flexible Integration of the Locking and Optimistic Approaches of Concurrency Control", Memorial University of Newfoundland, TR-8402, March 1984.

[Weihl85] William Weihl and Barabra Liskov, "Implementation of Resilient, Atomic Data Types", in *ACM Transactions on Programming Languages and Systems*, 7(2), April 1985.

[Wolfson87] Ouri Wolfson, "The Overhead of Locking (and Commit) Protocols in Distributed Databases", in *ACM Transactions on Database Systems*, 12(3), September 1987.