Logging and Recovery in a Highly Concurrent Database

by

John Sidney Keen

B.A.Sc., Electrical Engineering
University of Waterloo, 1986

M.A.Sc., Electrical Engineering
University of Waterloo, 1987

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Science

at the
Massachusetts Institute of Technology
May 1994

Signature of Author ______________________________

Certified by ______________________________

William J. Dally
Associate Professor of Electrical Engineering and Computer Science
Thesis Advisor

Accepted by ______________________________

Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Students

Logging and Recovery in a Highly Concurrent Database

by John Sidney Keen

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of Doctor of Science
at the Massachusetts Institute of Technology, May 1994

Abstract

This thesis addresses the problem of fault tolerance to *system failures* for database systems that are to run on highly concurrent computers. It assumes that, in general, an application may have a wide distribution in the lifetimes of its transactions.

Logging remains the method of choice for ensuring fault tolerance, but this thesis proposes new ways of managing a database's log information that are better suited for the conditions described above. The disk space reserved for log information is managed according to the *extended ephemeral logging (XEL)* method. XEL segments a log into a chain of fixed-size FIFO queues and performs generational garbage collection on records in the log. Log records that are no longer necessary for recovery purposes are "thrown away" when they reach the head of a queue; only records that are still needed for recovery are forwarded from the head of one queue to the tail of the next. XEL does not require checkpoints, permits fast recovery after a crash and is well suited for applications that have a wide distribution of transaction lifetimes. The cost of XEL is more main memory space and possibly a slight increase in the disk bandwidth required for log information. XEL can significantly reduce the disk bandwidth required for log information in a system that has been augmented with a non-volatile region of main memory.

When bandwidth requirements for log information demand an arbitrarily large collection of disks, they can be grouped into separate *log streams*. Each log stream consists of a small fixed number of disks and operates largely independently of the other streams. XEL manages the storage space of each log stream. Load balancing amongst the log streams is an important issue. This thesis evaluates and compares three different distribution policies for assigning log records to log streams.

Simulation results demonstrate the effectiveness of the implementation techniques proposed in this thesis for a highly concurrent database system in which transactions may have a wide distribution in lifetimes.

Thesis Advisor: William J. Dally
Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

This thesis reflects a life-long learning process. Many people have guided and encouraged me along the way, and they have my deepest gratitude. From my earliest days of childhood, my parents, Jim and Carolyn Keen, fostered my creativity and encouraged me to learn and think. When my love of technical engineering-style topics became apparent, they supported my further development in this direction. Thanks, Mom and Dad.

I am deeply indebted to my advisor here at MIT, Bill Dally. It has been a great privilege to work with Bill. His intellect, energy and enthusiasm are simply amazing. Inspirational! Bill graciously gave me considerable freedom to choose a research problem for my thesis, and then he helped me to follow it through to a finished thesis. Thanks for your encouragement, prodding and suggestions, Bill. I am also very grateful to the readers on my thesis committee: David Gifford, Nancy Lynch and Bill Weihl. They all offered many helpful comments and suggestions on preliminary drafts of my thesis.

My colleagues in the CVA (Concurrent VLSI Architecture) group were a pleasure to work with and they helped me to refine my ideas. Lots of good criticisms and valuable suggestions during presentations at group meetings assisted me in my work. The friendship of all CVAers created a wonderful environment. I thank my current officemates, Kathy Knobe and Diana Wong, for their cheerful company during this last hectic year.

During my time at MIT, I always lived at Ashdown House (most of the time on the fifth floor). I was fortunate to live in such a vibrant community of terrific people. There's not enough space to list everyone's name, but you know who you are and I thank you all for your kind friendship during our years together at MIT. I must make special mention of Vernon and Beth Ingram, who have been the housemasters at Ashdown throughout my stay. They dedicated themselves to making Ashdown a truly special place to live and I shall be forever grateful to them.

I thank God that I have been blessed with the opportunity and ability to study at MIT and complete this thesis. May the results of my work be to the glory of God and the good of society.

The research described in this thesis was supported in part by a Natural Sciences and Engineering Research Council of Canada (NSERC) 1967 Scholarship and in part by the Advanced Research Projects Agency under contracts N00014-88K-0738, N00014-91-J-1698 and F19628-92-C-0045 and in part by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation and AT&T Corporation. I thank all these sponsors for generously supporting my research.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

This thesis re-examines the problem of fault tolerance within the context of highly concurrent databases whose applications may be characterized by a wide distribution in transaction lifetimes. The goal of this effort is to propose and evaluate new data structures and algorithms that constitute an efficient and scalable solution to the fault tolerance problem. This thesis devotes most of its attention toward the management of information on disk and ignores other aspects of the problem. Current technical and economic trends justify this approach. Processors have made dramatic improvements, in terms of both cost and performance, compared to disk technology. Similarly, main memory storage space and interconnection network bandwidth are now relatively inexpensive (compared to disk) and abundant in most concurrent computer systems. Disk technology is becoming an increasingly crucial factor in the design of any concurrent database management system (DBMS) because it accounts for a significant fraction of the cost of a system and threatens to limit the system's performance [22, 7].

Highly concurrent computers offer the potential for powerful databases that can process many thousands of transactions per second. According to [18], a good database system typically requires 10^5 instructions per transaction for the debit-credit benchmark. Current microprocessors can process instructions at a rate of at least 10^8 instructions per second [29]. With current technology, therefore, it is reasonable to assume that each processor can support a rate of at least 1000 TPS if there are no limitations other than CPU speed. The overall performance of a system with hundreds of processors is expected to be several hundred thousand transactions per second for the debit-credit benchmark. A good DBMS design must eliminate bottlenecks that would otherwise prevent users from fully harnessing the computational potential of highly concurrent computers.

Data structures and algorithms that worked well in DBMS designs for serial computers are inappropriate for highly concurrent systems. Consider the specific problem of fault tolerance to *system failures* (also known as *crashes*), in which the contents of volatile main memory storage are corrupted. To provide support for atomic transactions, a DBMS must guarantee fault tolerance to crashes. Traditionally, DBMSs have kept a log of all modifications performed by transactions that are still in progress. Conceptually, the log is a FIFO queue to which records are added at the tail; the log should be sufficiently long that records become unnecessary before they reach the head. This abstraction of a single FIFO queue becomes awkward and inappropriate in a highly concurrent system; the tail of the queue is a potential serial bottleneck. Furthermore, if only a single disk drive is dedicated to hold the log, it may not provide sufficient bandwidth for large volumes of log information. For example, a system that generates 500 Bytes of log information per transaction and runs at 100,000 TPS needs at least 50 MBytes/sec of disk bandwidth. If current disk drive technology can provide at most 2 MBytes/sec bandwidth per drive, the system requires at least 25 disk drives just for log information to ensure that the log is not a bottleneck.

To add to these difficulties, applications that use databases are becoming more diverse. Some applications may have a wide distribution of transaction lifetimes. An application with a small proportion of transactions whose lifetimes are much longer than average poses problems for traditional logging algorithms. Most variations of logging retain all log records that have been written (by all transactions) since the beginning of the oldest transaction that is still in progress. Many of these log records may be unnecessary for the purposes of recovery, but their disk space cannot be reclaimed as long as some older record must be retained; this situation arises as a consequence of the FIFO policy which governs the management of a log's disk space. This constraint poses disk management problems. If a transaction lives too long, the log will run out of space to hold new records. An obvious solution is to simply allocate a large amount of disk space for the log, but this implies some unpleasant consequences. First, it may unnecessarily increase a system's cost. Second, the large size of the log may entail a much longer recovery time after a crash. These drawbacks prompt an investigation into better methods of fault tolerance for databases whose applications may have a wide distribution in transaction lifetimes.

1.2 Major Contributions

The following paragraphs summarize the major contributions of this thesis.

Extended Ephemeral Logging (XEL). XEL is a new technique for managing a log of database activity on disk. This thesis presents the data structures and algorithms which constitute XEL; it explains XEL's operation during both normal database activity and recovery from a crash. XEL does not require periodic checkpoints and does not abort lengthy transactions as frequently as traditional logging techniques which manage the log as a FIFO queue (assuming that XEL and the FIFO queue technique are both limited to the same amount of disk space). Therefore, XEL is well suited for highly concurrent databases and applications that have a wide distribution of transaction lifetimes. XEL can offer significant savings in disk space, at the expense of slightly higher bandwidth for log information and more main memory. The reduced size of the log permits much faster recovery after a crash as well as cost savings. XEL can significantly reduce both disk space and disk bandwidth in a system that has at least some portion of main memory which is non-volatile.

Proof of Correctness for XEL. XEL's safety and liveness properties are formally proven. Apropos safety, this thesis proves that XEL never does anything wrong; therefore, the database can always be restored to a consistent state after a crash, regardless of when the crash occurs. The liveness property ensures that XEL always makes progress; every log record is eventually erased.

Evaluation of XEL. The benefits and costs of XEL, relative to logging techniques which manage log information in a FIFO queue, are quantitatively evaluated via event-driven simulation. This thesis presents these experimental results.

Evaluation of Parallel Logging Distribution Policies. The abstraction of multiple *log streams* for log information can be easily implemented in a highly concurrent DBMS which requires an arbitrarily large collection of disk drives to provide the necessary bandwidth for log information. A database system's *distribution policy* dictates the log stream(s) to which any particular log record is sent. This thesis evaluates and compares three different distribution policies for a DBMS that has multiple parallel streams of log records. The *random* policy, which randomly chooses a log stream for any log record, has good load balancing properties and is simple to implement.

Logged Commit Dependencies (LCD). The LCD technique permits very high throughput on "hot spot" objects in a highly concurrent database. It is a variant of the precommitted transaction technique [15] and is especially well suited for a DBMS that uses multiple parallel log streams. A transaction's dependencies on previous precommitted transactions are explicitly encoded in a special `PRECOMMIT` record that is generated as soon as the transaction requests to commit, thus eliminating potentially awkward synchronization requirements between the log stream to which the transaction's `COMMIT` record is eventually written and the log streams to which `COMMIT` records are written for the transactions on which it depends.

1.3 Statement of Problem

In any DBMS, the *log manager (LM)* manages log information during normal database operation, and the *recovery manager (RM)* is responsible for restoring the database to a consistent state after a crash. Together, these two components make up a DBMS's logging and recovery subsystem. The log holds records for only recent modifications to the database. A version of the database kept elsewhere on disk stores the state of all items of data in the database. At any given point in time, this disk version of the database does not necessarily incorporate all the updates that have been performed by committed transactions; some of these updates may be recorded only in the log. Another DBMS component called the *cache manager (CM)* is responsible for managing the contents of the disk version of the database and must work in collaboration with the LM. The CM chooses to *flush* (transfer) updated objects[1] to the disk version of the database in a manner that uses I/O resources efficiently.

Figure 1.1 graphically represents the disk configuration of a concurrent database system. At the top, a collection of disk drives provide the bandwidth and storage capacity required for log information generated by the DBMS; these are called the *log disks*. The LM manages these drives. The exact number of log disks is chosen to support the highest rate of transaction processing of which the rest of the system is capable. A small number of buffers, in main memory, are dedicated to each of these log disks. On the right hand side, some other collection of disk drives hold the disk version of the database. The exact number of drives required for the disk version of the database depends on the demands for disk space and bandwidth imposed by the rest of the system. A buffer pool is associated with each different disk drive of the disk version of the database. These buffer pools are quite large so that they serve as caches. They reduce the number of retrievals from disk and allow writes to disk to be ordered in a manner that permits higher transfer rates (due to mostly sequential I/O). The CM manages these buffer pools and their associated disk drives.

A complete solution to the logging and recovery problem in a concurrent DBMS must answer all of the following questions, which apply to the management of log information during normal operation of the database:

1. What events are logged?
2. What information should a log record contain?
3. How does the LM decide the disk drive(s) to which it will write a log record?
4. At what time should the LM write a log record to disk?
5. Where on disk should the LM write a log record?

[1]The term *object* is used broadly to denote any distinct item of data in a database. It may be a record in a hierarchical or network database, a tuple in a relational database or an object in an object-oriented database.

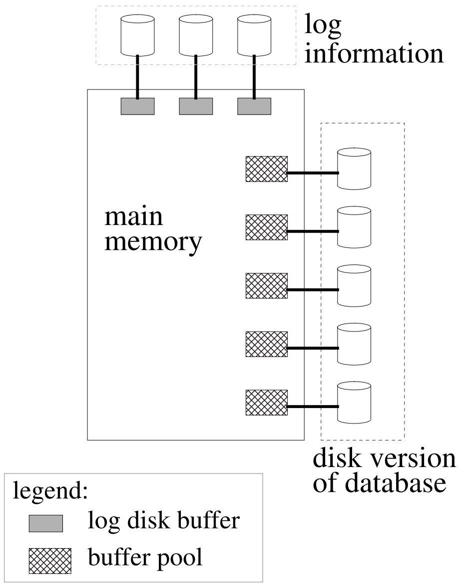


Figure 1.1: Disk Configuration for Concurrent Database System

6. When can the LM overwrite a log record on disk with more recent log information?
7. When can the CM flush an updated object's new value to the disk version of the database?

Any proposed logging and recovery method must also respond to the following questions that concern recovery after a crash:

1. How should the RM schedule retrievals of blocks of log information from disk?
2. In what order does the RM process the log records contained in a block of log information?
3. Given a particular log record, what does the RM do with it?

1.4 Review of Previous Research

Several good textbooks and articles have been published on the subject of fault tolerance in database systems. A reader who would like to become familiar with the basic techniques and terminology of the field is referred to [20, 24, 30, 5, 26]. The remaining subsections of this section review prior research that is specifically relevant to the material in this thesis.

1.4.1 Disk Storage Management

The Firewall Method of Disk Management

Traditionally, logging has been the method of choice for fault tolerance in most database systems[2]. A LM maintains a log of database activity as transactions execute and modify items of data (i.e., objects) in the database. Conceptually, the log is a FIFO queue. The LM adds log records to the tail immediately after they are created. Log records progress toward the head of the queue and ought to become unnecessary by the time they eventually reach the head. The DBMS allocates a fixed amount of disk space to hold log information. The LM manages this disk space as a circular array [3, 10]; the log's head and tail pointers rotate through the positions of the array so that records conceptually move from tail to head but physically remain in the same place on disk. System R [24] is a familiar example of this traditional logging technique.

The LM maintains a pointer to the oldest record in the log that must still be retained; this constitutes a "firewall" beyond which the head of the log cannot be advanced. Hence, this logging technique shall be referred to as the *firewall (FW)* method. The LM initiates periodic checkpoints. As soon as a checkpoint begins, the LM writes out a special *beginning-of-checkpoint* record to the log. During a checkpoint, the CM writes out all updated objects to the disk version of the database[3] and then the LM writes out a special *end-of-checkpoint* record to the log. After the checkpoint has completed, the LM can be sure that all preceding log records for committed updates are no longer necessary to ensure correct recovery of the database after a crash. The LM keeps a pointer to the position within the log of the *beginning-of-checkpoint* record for the most recently completed checkpoint. The LM also maintains a pointer for each active[4] transaction that identifies the position within the log of the oldest record written by the transaction. At any given time, the log's firewall is the oldest of the pointers for all active transactions and the pointer to the beginning of the most recent checkpoint. Figure 1.2 illustrates an example.

If the log starts to run short on space for new log records, the LM must free up some space by advancing the firewall. It must either kill an old active transaction or it must perform another checkpoint, depending on the exact nature of the current firewall. In general, it is bad to kill a transaction because this will likely annoy the client who originally initiated the transaction. Furthermore, all the resources consumed by the transaction have essentially been wasted, and the transaction's effort will be repeated

[2]Logging is not the only possible solution to the fault tolerance problem. However, it has tended to be the most popular solution for reasons of performance and efficiency. Refer to [37, 30, 2, 5, 39] for explanations of alternative methods of achieving fault tolerance (such as shadowing) and comparisons of the strengths and weaknesses of the different approaches.

[3]Sophisticated *fuzzy* checkpoint methods allow the database to continue servicing requests from client transactions while the CM flushes out all updated objects, so that the checkpoint activity causes negligibly small disruption to normal operation of the database.

[4]An *active* transaction is one that is still in progress (it has not committed nor been aborted).

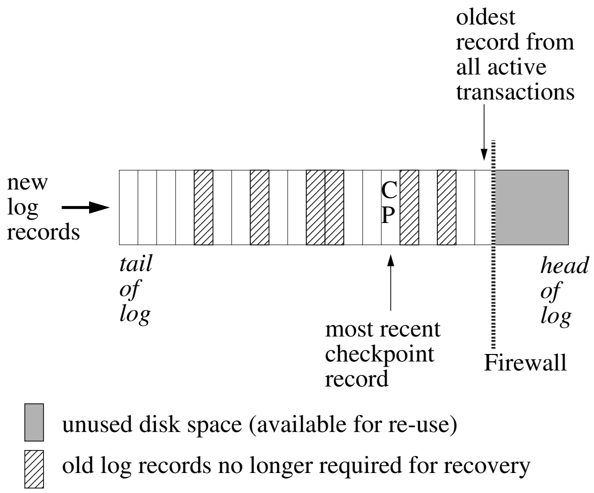


Figure 1.2: Firewall Method of Disk Space Management for Log

if its client decides to try it again. This scenario is particularly irritating because many records in the log may be unnecessary for recovery purposes but the FIFO queue abstraction prevents the LM from reclaiming their space until all prior log records have been rendered unnecessary. For example, suppose that a transaction updates an object and continues to live for another 10 min while many short (several seconds) transactions each update a few objects and commit. The log records from these short transactions follow the long transaction's first log record. Even though most of the log records from these short transactions are no longer needed for recovery, their space cannot be reclaimed until the long-lived transaction finishes.

Checkpoints are not free. They become awkward in a concurrent system because they entail synchronization and coordination amongst an arbitrarily large number of participating parties. In general, if the LM wishes to perform a checkpoint, it must coordinate activity at all the log disks and all the disk drives on which the disk version of the database resides. A checkpoint operation requires communication bandwidth, processor cycles, storage space in main memory and disk bandwidth. Periodic checkpoints may interfere with the CM's operation by constraining it to schedule flushes to disk in an order that does not take full advantage of locality within the disk version of the database. Finally, the duration required to perform a checkpoint and the delays between consecutive checkpoints limit the speed with which the LM can reclaim space in the log.

As the number of log disks increases, the abstraction of a single queue becomes increasingly difficult to implement. The tail of the queue is a potential serial bottleneck and the LM must carefully manage the order in which it writes blocks to each log disk so that it preserves the log's FIFO property.

Therefore, the traditional "firewall" method of logging poses implementation problems for highly concurrent databases with wide variations in transaction lifetimes because it does not reclaim disk space sufficiently quickly, it suffers from the overhead of periodic checkpoint operations and it is difficult to implement on an arbitrarily large number of disk drives.

Recirculation of Log Records

Hagmann and Garcia-Molina [32] propose a solution to the disk management problem posed by long lived transactions. If a record reaches the head of the log but must still be retained, the LM recirculates the record within the log by adding it to the tail.

In contrast to the XEL method which this thesis will propose, they continue to implement the log as a single FIFO queue, rather than a chain of FIFO queues. A log record from a very long lived transaction may therefore be recirculated numerous times within the log, thereby consuming more bandwidth than would be required by the XEL method. Furthermore, Hagmann and Garcia-Molina do not attempt to eliminate the need for checkpoint operations.

Log Compression

Log compression filters out unnecessary information so that less storage space is required to store the log. However, previous methods of log compression [34, 31] are intended for "batch mode" log compression; the LM cannot add new records to the existing log while it is being compressed.

The XEL method proposed in this thesis essentially performs log compression, but its continuous and incremental nature distinguishes it from previous log compression methods.

Generational Garbage Collection

Previous work on generational garbage collection inspired XEL's essential idea: the log is segmented into a chain of FIFO queues. Lieberman and Hewitt [40] proposed the segmentation of a system's main memory storage space into several temporal generations; most of the system's garbage collection effort is limited to only its younger generations. Quantitative evaluation of several variations on generational garbage collection [62, 49, 63, 59] have demonstrated its effectiveness for automatic memory management.

However, previous work on generational garbage collection addressed the more general problem of automatic storage reclamation by a programming language's runtime system. The reference pattern amongst a program's data objects is usually more complicated than the dependencies that exist amongst log records, and so garbage collection methods that worked well for programming languages may be inappropriate for managing the disk storage reserved for a database's log. Less complicated yet more effective techniques may be possible for the simpler problem of managing a database's log.

Rosenblum and Ousterhout [56, 57, 58] adopt a similar strategy for the log-structured file system (LFS). The LFS adds all changes to data, directories and metadata to the end of an append-only log so that it can take advantage of sequential disk I/O. The log consists of several large segments. A segment is written and garbage collected ("cleaned") all at once. To reclaim disk space, the LFS merges non-garbage pieces from several segments into a new segment; it must read the contents of a segment from disk to decide what is garbage and what isn't. There is a separate checkpoint area and the LFS performs periodic checkpoints. The LFS takes advantage of the known hierarchical reference patterns amongst the blocks of the file system during logging and recovery; the *inode map*, *segment summary blocks* and *segment usage array* data structures describe the file system's structure and play an important role in the LFS's management of disk space.

1.4.2 Parallel Logging and Recovery

Distributed databases are similar to concurrent databases, but not identical. Like concurrent databases, a distributed database consists of an arbitrary number of processors linked via some communication network; the total set of processors is partitioned into disjoint *sites*, and the communication network connects the sites together. These sites can share data, so that a transaction executing at any particular site effectively sees one large database. However, the underlying technology distinguishes distributed databases from concurrent databases. In general, a distributed database's communication network has lower bandwidth, much longer latency and lower reliability than that of a concurrent database. Moreover, individual sites or links in a distributed database's network may fail (*partial failures*, in the terminology of [5]), yet other portions of the system remain intact; the system can continue to provide service to client transactions as long as these transactions do not require access to data that is unavailable due to failures elsewhere. For many concurrent systems, either the entire system is operational so that all data is available to client transactions or it is completely failed (a *total failure* [5]) so that no transaction can execute. This "all or nothing" characteristic simplifies some aspects of the DBMS implementation problem.

The limitations of a distributed database's communication network and concerns about availability of data lead to rigid partitioning of data objects within the system. Assume, for simplicity, that objects are not replicated at different sites. Each object has a *home site* [5] and objects do not migrate between sites. Any transaction that wants to access a particular object must do so at its home site. Whenever an object is updated, log information is generated and stored at its home site. This rigid partitioning is liable to load balancing problems. One site may hold a disproportionately large number of "hot spot" objects[5] and so it is overloaded while another site is relatively idle. However, the rigid partitioning prevents the system from taking advantage of available processing power and disk bandwidth elsewhere in the system. A concurrent system has more flexibility in balancing the loads amongst processors and disk drives. The possibility of partial failures necessitates complicated algorithms for atomic commitment, such as the *two phase commmit (2PC)* and *three phase commit (3PC)* protocols [5], for distributed databases. These sophisticated techniques are unnecessary for concurrent databases under the "all or nothing" assumption.

The SWALLOW distributed storage system [54] stores objects in a collection of autonomous *repositories*. Each object is represented as a sequence of versions, called a *version history*, which records successive updates to the object over time. SWALLOW creates a commit record at each *repository* at which a transaction updates objects and links new versions of updated objects to the transaction's commit record while the transaction executes. If the transaction eventually aborts, the system deletes the transaction's commit record and the versions for the updates which it performed on objects. Otherwise, the committed transaction's changes become permanent in the multiversion representation of the database. Hence, there are no distinct log records in SWALLOW because it retains versions for objects. Old versions can be garbage-collected when there are no longer any references to them.

Lomet [41] proposes a new method for redo logging and recovery that is intended for use in a data sharing system. Multiple nodes can access common data, yet each node can have its own log. In contrast to the partitioning of data which characterizes many distributed databases, Lomet's system allows data objects to migrate within the system. After modifying an object, a processing node records the operation in its own private log; each private log is a sequential file, apparently managed as a FIFO queue. Hence, log records for different updates to the same object may be distributed throughout a collection of private logs. This data sharing approach offers potentially better load balancing behavior. For each object, Lomet's system ensures that at most one node's log can record updates that have not yet been applied to the version of the object currently in the disk version of the database; hence, recovery activity for each object is still limited to only a single node even though an object's log records may be distributed throughout the private logs of numerous nodes.

Previous researchers have already broached the problem of parallel logging and recovery in a concurrent database but they often focussed on only isolated subproblems or declined to propose detailed solutions. The expedient of using several disk drives to increase the throughput for log information was suggested in [15]. Lehman and Carey [39] propose storing log information in a logically parallel manner that easily maps to a physically parallel implementation; every *partition*[6] in the database has its own separate

[5] "Hot spot" objects are accessed much more frequently than other objects in the database.

[6] A *partition*, as defined in [39], is the unit of memory allocation for a system's underlying memory

log of activity.

Agrawal [1, 2] investigates some alternative solutions to the recovery problem for the *multiprocessor-cache* class of database machines, of which the DIRECT system [14, 8] is an example. He investigates four different distribution policies for determining the log disk to which to send a log record: *cyclic*, *random*, *Query Processor Number mod Total Log Processors (QPNmTLP)* and *Transaction Number mod Total Log Processors (TNmTLP)*. Agrawal concluded that the TNmTLP policy suffers from significant load balancing problems; an exceptionally industrious transaction can generate a deluge of log information for one log disk while the other disks sit relatively idle. As processor speed and bandwidth continue to rise, relative to disk bandwidth, the QPNmTLP policy faces similar load balancing problems. Agrawal examined these four policies within the context of the multiprocessor-cache class of database machines and in conjunction with different solutions to some of the other subproblems associated with logging and recovery. His evaluation criteria focused on overall performance (throughput and response time) rather than on specific aspects of efficiency. He does not consider how much disk space the log requires or the extent to which the log disks' loads are balanced. This thesis will make different assumptions that more closely model today's concurrent computer technology and will choose different evaluation criteria.

Apropos recovery, DeWitt *et al.* [15] propose merging several parallel logs into a single log so that familiar algorithms from the sequential world will be applicable. Agrawal's algorithm [1] does not require a merge step, but it processes each log sequentially one after the other and therefore forfeits the opportunity to exploit parallelism during recovery. Kumar [38] proposes a parallel recovery algorithm, but it requires processing at all log streams to proceed in a lock-step manner; frequent barrier synchronization limits the performance and scalability of this algorithm. To address these limitations, he proposes an improved algorithm that involves minimal synchronization between recovery activities at separate log streams, but this latter algorithm still requires two scans over each log stream.

1.4.3 Management of Precommitted Transactions

Interactions between the LM and the concurrency control manager (CCM) of a DBMS can affect a system's overall performance. *Strict two phase locking (2PL)* [5] requires that the CCM release all write locks held by a transaction only after the transaction has committed or aborted. Hence, the CCM must hold all the write locks of a successful transaction for at least as long as the minimum response time for the LM to accept a DLR and process a request to commit from the transaction. Without non-volatile main memory, the lower bound for this response time is the minimum time required to write a block to disk. Slow response on the part of the LM may entail concurrency control bottlenecks on some objects that must be updated *very* frequently.

mapping hardware.

To alleviate these performance limitations, the *precommitted transaction (PT)* technique [15] enables a CCM to release a transaction's write locks as soon as the transaction requests to terminate. The transaction *precommits* when it requests to commit, and it *commits* when all its log records (including a COMMIT record) have been written to disk. It is in a *precommitted* state between these two times.

The PT technique alleviates the throughput bottleneck on "hot spot" objects due to I/O latency to disk. Without the PT technique, a transaction that has requested to commit cannot release its write locks until after it has committed, lest some other transaction see its effects before they have been made permanent in the database. In this case, the maximum rate at which independent transactions can update an object is limited by the rate at which successive blocks of log records can be written to disk. If the minimum time to write a block to disk is τ_{min} (for example, 10 ms), then each transaction must hold its write locks for at least τ_{min} and the maximum throughput for any object is $1/\tau_{min}$. If a database has hot spot objects, its entire throughput may therefore be limited to $1/\tau_{min}$. When the LM uses the PT technique, independent transactions can update hot spot objects at a much higher rate, limited by the time required to acquire and release locks.

Now suppose that a DBMS's LM supports precommitted transactions. A transaction can release its write locks after it has requested to commit but before it actually commits. While it waits in this precommitted state, the only thing that could cause it to fail is some failure on the part of the DBMS (e.g., a crash) which prevents the log records from being written to disk. Other transactions can see the updates from the precommitted transaction, but they become *dependent* on it to eventually commit. The LM sends an acknowledgement to the transaction in response to its commit request after the transaction commits. If a crash occurs before the precommitted transaction commits, the RM must ensure that the restored database does not include updates from the transaction or any of the subsequent transactions which depended on it. After a transaction commits, a COMMIT log record exists on disk to record the fact that the transaction committed and so its effects are guaranteed to survive a crash.

1.5 Commercial Systems

This section briefly reviews existing commercial database systems. Most commercial systems incorporate variations of the techniques presented in the previous section.

IBM has had a long and influential presence in the database market. Details about several of its most noteworthy products can be found in [26]. IMS, one of the industry's earliest database systems, runs on the MVS operating system; MVS runs on computer systems built around the IBM 370 processor family. Early versions of IMS maintained separate redo and undo logs. Redo information was kept for restart recovery and undo information (the *dynamic log*) was for transaction backout. More recently, IMS has merged the redo and undo logs into one log to reduce I/O at commit. IMS supports group commit [5] and performs fuzzy dumps for archive recovery. IBM's IMS FastPath system introduced the notions of group commit and main-storage databases, among other things. It is a pure deferred-update, redo-only system (this implies a ¬STEAL buffer management policy [5]). DB2 is IBM's implementation of SQL for its mainframe systems; it implements a ¬STEAL buffer management policy and the WAL (write ahead log) protocol [5].

DEC markets a database system called Rdb/VMS. It runs on a VAXcluster system ("shared disk" architecture) and supports a single global log. Rdb/VMS performs undo/redo logging, with separate journals for redo and undo log records; the *After Image Journal (AIJ)* holds redo information and the *Run-Unit Journal (RUJ)* keeps undo information. Periodic checkpoints bound the length of recovery time. Rdb/VMS exploits group commit to achieve efficient disk I/O.

The Teradata DBC/1012 system [60, 46, 47, 55] is a highly parallel database system that is intended principally for decision support (i.e., mostly queries) but which provides some support for on-line transaction processing (OLTP). The most recent version, the DBC/1012 model 4, incorporates up to 1024 Intel 80486 microprocessors. The system's processors are divided into *interface processors (IFPs)* and *access module processors (AMPs)*. The DBC/1012 can support multiple hosts; each IFP connects to one particular host. The AMPs manage the data. Tuples within a relation are partitioned across the AMPs so that the DBC/1012 can support parallelism both within and between independent requests. The AMPs and IFPs are interconnected by a proprietary Ynet "active logic" interconnection network.

The Tandem NonStop SQL system [27, 28, 33, 16] is essentially a distributed relational database system. Data objects are partitioned across multiple processing nodes and transactions can access data at different sites. The system provides local autonomy so that a site can perform work despite failures at other sites or in the interconnection network. An implementation of the two-phase commit (2PC) protocol [5] ensures that distributed transactions commit atomically. NonStop SQL performs undo/redo logging and maintains a separate log at each site. The state of the database is periodically archived by performing a fuzzy dump.

Oracle's Parallel Server [44] is intended to run on highly parallel systems (such as the KSR1 [61]). It can support very high transaction processing rates, compared to other available systems. Version 6.2 has been benchmarked at 1073 TPS (transactions per second) for the TPC-B benchmark [21], with a cost of only $2,480 per TPS; these results were obtained for Oracle V6.2 running on an nCube concurrent computer system with 64 processors. Parallel Server employs redo/undo servers for log information, performs periodic checkpoints and uses a partitioned distribution policy for distributing log records.

1.6 Assumptions

Physical State Logging at the Access Path Level. This thesis limits its attention to physical state logging at the access path level [30]. According to this definition, any modification to an object in the database results in a log record that holds some representation of the state of the object; the log record may hold the pre-modification state of the object, the post-modification state, or both.

Buffer Management: ¬FORCE, STEAL. This thesis assumes the most general policy for buffer management. The CM may flush an updated object to the disk version of the database whenever it chooses, regardless of whether or not the transaction that performed the update has yet committed. Restated in formal terminology, the buffer management policy is ¬FORCE and STEAL [30].

Concurrency Control: Two Phase Locking. The LM does not perform concurrency control, but the concurrency control manager that schedules requests to the LM on behalf of client transactions must respect certain restrictions if the RM is to be able to restore the database to a consistent state after a crash. This thesis assumes that the concurrency control manager performs *two phase locking (2PL)* [17, 5] so that all executions are *serializable*.

All chapters except Chapter 5 will further assume that all executions are *strict* [5]: no transaction reads or updates an object that has been modified by another transaction which has not yet committed. Chapter 5 relaxes this assumption slightly; a transaction may release its write locks shortly after it requests to commit even though it has not actually committed yet.

Volatile Main Memory, Non-volatile Disk Storage. All main memory is volatile. In the event of a *system failure*, such as a power interruption, some or all of the contents of main memory may be lost. In contrast, disk storage (secondary memory) is non-volatile. Any information written to disk will remain on disk despite a system failure.

Distributed Memory Multiprocessor System. The data structures and algorithms presented in this thesis are intended for a fine-grain distributed memory multiprocessor system, such as the MIT J-Machine [11, 12, 48], in which each processor can directly address only its own local memory and all interprocessor communication must occur via explicit message passing. Nevertheless, the techniques presented in this thesis could be adapted to a shared memory multiprocessor system with little effort.

Disks are the Limiting Resource. This thesis addresses the problem of managing log information on disk, subject to limited storage capacity and bandwidth. Disk technology threatens to limit the performance of concurrent database systems, and is expected to account for a significant fraction of their cost [22, 7]. Existing concurrent computer systems provide abundant computational power, volatile main memory storage and interprocessor communication ability so that none of these resources constitutes a bottleneck. Hence, the attention specifically to disk technology.

Recent trends and expectations for future progress justify this assumption. Processor performance has increased dramatically over the past decade and will likely continue to improve at a fast pace in the near future. Similarly, DRAM (dynamic random access memory) capacities have soared and prices have fallen during the past decade, and these trends in main memory technology are expected to continue. Interconnection network technology has improved significantly so that high bandwidth, low latency interprocessor communication is now a reality. For example, the MIT J-Machine provides 288 Mbps communication bandwidth per channel [12, 50], and each processing node has 6 channels. In contrast, the capacity, bandwidth and cost of disk drives have not improved as dramatically.

Unique Identifiers for Objects and Transactions. Every object in the database must have some unique *object identifier (oid)*. Similarly, each transaction must have a *transaction identifier (tid)* that distinguishes it from all other transactions.

1.7 Summary of Remaining Chapters

Chapter 2 explains the extended ephemeral logging (XEL) method. XEL is a new method for managing a log of database activity on disk. It is a more general variation of ephemeral logging (EL) [35]; XEL does not require a timestamp to be maintained with each object in the database. XEL does not require periodic checkpoints and does not abort lengthy transactions as frequently as traditional firewall logging for the same amount of disk space. Therefore, it is well suited for highly concurrent databases and applications that have a wide distribution of transaction lifetimes.

Important safety and liveness properties for a simplified version of XEL are proven in Chapter 3. The log record from the most recently committed update to an object remains recoverable as long as log records from earlier updates to the same object can be recovered from the log. However, every log record is eventually erased so that its space on disk can be re-used for subsequent log information.

Chapter 4 considers how to manage log information in a highly concurrent database. The abstraction of a collection of *log streams*, all operating in parallel with one another, is suitable for applications with very high bandwidth requirements for log information. The LM uses the XEL method to manage the disk space within each log stream. With multiple log streams, the LM must have some distribution policy by which it decides the stream(s) to which it will send any particular log record. Chapter 4 analyzes three different distribution policies.

Chapter 5 points out the difficulties of implementing the PT technique, in its current form, in a LM that supports an arbitrarily large collection of log streams. The chapter proposes a new variation of the technique, called *Logged Commit Dependencies (LCD)*, which alleviates these difficulties. It introduces a new type of record, called a `PRECOMMIT` record, which explicitly states all a transaction's dependencies at the time that the transaction requests to commit.

Chapter 6 quantitatively evaluates XEL via event-driven simulation. XEL's complexity severely limits analytical attempts to evaluate its performance. Simulation provides an alternative means by which to study its behavior. Section 6.1 describes the implementation of a simulator for XEL. It explains each of the input parameters, documents the fixed parameters, presents the definitions of XEL's data structures as expressed in the C programming languange [36] and justifies the validity of the simulation model. Section 6.2 evaluates XEL's performance for only a single log stream as various input parameters vary and compares XEL's performance to that of the FW method. The following section examines XEL's behavior for a collection of parallel log streams as the degree of parallelism increases. Disk space, disk bandwidth, main memory requirements and recovery time are the evaluation criteria throughout the chapter.

The last chapter of the thesis summarizes the important lessons that were learned, explains the importance of the results and discusses various extensions to XEL.

Chapter 2

Extended Ephemeral Logging (XEL)

This chapter proposes a new technique for managing the disk space allocated for log information. This new technique, called *extended ephemeral logging (XEL)*, is a more general variation of the ephemeral logging (EL) technique that the author presented in an earlier publication [35]. Both EL and XEL break the abstraction of the single FIFO queue that was presented in Section 1.4.1. Rather than managing log information in a single FIFO queue, EL and XEL treat the log as a chain of fixed-size FIFO queues and perform garbage collection at the head of each queue. This approach, inspired by previous research on generational garbage collection, mitigates the threat of the log running out of space for new log records because a transaction lives too long; EL and XEL can retain the records from long running transactions but can reclaim the space of chronologically subsequent log records that are no longer needed for recovery. Hence, a log manager that uses EL or XEL generally requires less disk space than one that treats the log as a single FIFO queue. Intuitively, this advantage is strongest if an application has only a small fraction of transactions that execute for a very long time and write only a small number of records to the log.

Another strong motivation for EL and XEL arises if a system can be augmented with a limited amount of non-volatile main memory. In such a system, EL and XEL can drastically reduce the amounts of disk space and bandwidth required for log information if most log records emanate from short-lived transactions. The benefits here are twofold. First, the system's cost may be substantially reduced since fewer disk drives are needed for log information. Second, recovery after a crash may be much faster since the amount of log information is considerably smaller.

Variations on EL and XEL can render a separate disk version of the database unnecessary. The most recently committed value for each object is always retained in the log.

This approach may significantly reduce a system's cost because it likely requires fewer disk drives. It also simplifies the DBMS design because the CM is no longer needed. This expedient pertains to main memory database systems as well as other systems which hold most of their data in main memory and update them sufficiently often.

EL and XEL maintain pointers to all records in the log that are relevant for recovery purposes. This entails significantly higher main memory requirements, compared to the FW technique. However, the LM no longer needs to perform periodic checkpoints to ensure that all updates prior to a particular point in time have been flushed to the disk version of the database, as had been necessary for the FW method. Of course, the CM should continue to flush committed updates to the disk version of the database at as fast a rate as possible so as to reduce the amount of information that must be kept in the log. This elimination of checkpoints is a benefit for highly concurrent systems which have many processors and an arbitrary number of parallel log streams (as will be discussed in Chapter 4). Checkpointing is more complicated in concurrent systems, compared to sequential systems, so EL and XEL relieve concurrent DBMS designers from having to design and implement efficient checkpointing algorithms that are provably correct.

The presence or absence of timestamps in the disk version of the database differentiates EL and XEL. EL assumes that each object's representation in the disk version of the database has a timestamp kept with it. However, there are good reasons why some databases may violate this assumption. The absence of timestamps complicates the problem of managing log information in a manner that does not jeopardize the consistency of the database. The XEL technique presented in this chapter does not require timestamps for objects in the disk version of the database.

Section 2.1 illustrates why EL cannot guarantee consistency after a crash for a DBMS that does not maintain a timestamp with every object in the disk version of the database. Once familiar with the pitfalls of EL, a reader will be better able to appreciate the rationale that underlies the complexities of XEL. Subsequent sections each address specific problems that must be solved in order to implement XEL. To some extent, these subproblems are independent of one another. The structure of this chapter reflects the modular nature of these problems.

2.1 Preamble: Limitation of Ephemeral Logging

Ephemeral logging (EL), as originally proposed in [35], adopts a generational garbage collection strategy for managing a log's disk space. The LM manages the log as a chain of FIFO queues, each of which is called a *generation*. The LM adds new log records to the tail of the youngest generation. When a record that must still be kept in the log approaches the head of a generation, the LM forwards it to the tail of the next generation or recirculates it within the last generation. For any particular application, the generations' sizes are chosen so that only a small fraction of the records in any generation need to be forwarded or recirculated.

EL requires each object in the database to have a monotonically increasing timestamp. The simplest implementation that satisfies this constraint maintains an integer-valued counter with every object. The LM increments an object's counter each time a transaction updates the object. Whenever the CM flushes an updated object to the disk version of the database, the accompanying timestamp value is stored with the object. Likewise, each *data log record (DLR)* for an object holds the value and corresponding timestamp (as well as the object and transaction identifiers) from a particular update to the object. After a crash, the RM can determine if the disk version of the database holds the most recently committed value for any particular object. It finds the most recently committed DLR for the object that is still in the log and checks if this DLR has a more recent timestamp than the version of the object currently on disk. If the DLR is more recent, then the RM should update the object in the disk version of the database; otherwise, it should ignore the DLR.

Now suppose that timestamps are not kept with each object stored in the version of the database on disk. This case might arise because of a deliberate decision to conserve storage, or it could be a constraint inherited from an existing implementation. Without timestamps in the database, EL is not sufficient to guarantee a consistent state after recovery from a crash. The RM no longer has a standard by which to judge whether or not a DLR holds a more recent value than that which currently exists for the corresponding object on disk. Accordingly, the RM can no longer deduce which records are non-garbage and which are garbage.

The following example illustrates what can go wrong. Assume that the log has two generations. Suppose transaction *tx3* assigns object *ob8* a value of 12, writes a corresponding DLR to the log and then commits. Assume that the DLR for this update and the transaction's `COMMIT` TLR are both forwarded to generation 1 of the log[1]. After moving to generation 1, these two records soon become garbage. Now suppose that transaction *tx6* subsequently assigns object *ob8* a value of 9 and then commits. Figure 2.1 summarizes this chronology of events for transactions *tx3* and *tx6*.

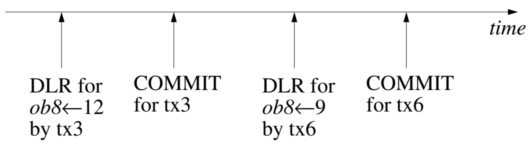


Figure 2.1: Two Successive Updates to Object *ob8*

Suppose further that both the DLR and the `COMMIT` TLR from *tx6* become garbage before they reach the head of generation 0 and are overwritten by other log records, but

[1]The DLR may have been forwarded because *tx3* had a relatively long lifetime, for example. The `COMMIT` TLR was forwarded because not all the transaction's updates had been flushed to the disk version of the database before it reached the head of generation 0.

the “stale” log records from *tx3* are still lingering in generation 1, as shown in Figure 2.2. If a crash were to occur while the system is in such a state, the RM would find *tx3*'s DLR to be the most recently committed update in the log when it attempts to recover object *ob8* but it would not know whether the value in this DLR is more recent than the value currently stored for *ob8* in the disk version of the database.

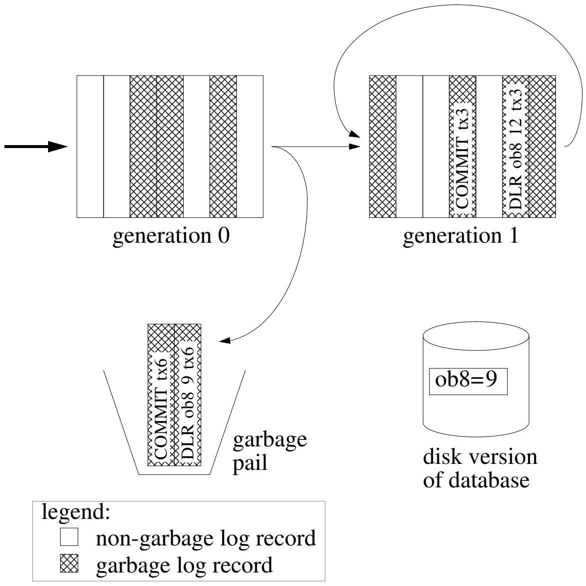


Figure 2.2: State of the Log After a Crash

In this example, the DLR from *tx3* is garbage and ought to be ignored; the disk version of the database already holds the most recently committed value for object *ob8*. It is not difficult to construct a different example in which the RM finds a (non-garbage) DLR that is more recent than the value stored for an object in the disk version of the database.

2.2 Conceptual Design of XEL

Extended ephemeral logging (XEL) manages a log's disk space as a chain of fixed-size queues. Each queue is called a *generation*. If there are N generations, then generation 0 is the *youngest* generation and generation $N-1$ is the *oldest* generation. New log records are added to the tail of generation 0. A log record near the head of generation i, for $i<N-1$, is forwarded to the tail of generation $i+1$ if it must be retained in the log; otherwise, it is simply discarded (overwritten by more recent log information). In the special case of generation $N-1$, a log record near its head that must be retained is recirculated in it by adding the record to its tail. The disk space within each queue is managed as a circular array [10]; the head and tail pointers rotate through the positions of the array so that records conceptually move from tail to head but physically remain

in the same place on disk.

In tandem with the activity of the LM, the CM flushes (transfers) updates to the disk version of the database so that some log records become unnecessary for recovery. The LM no longer needs to retain these log records and so it can re-use their space on disk.

Figure 2.3 conveys the essence of XEL for the specific case of a log stream with three generations. *Non-garbage* log records are necessary for recovery and must be kept in the log; all other log records are *garbage*. The "garbage pail" does not actually exist, but is conceptually convenient to suggest that log records are "thrown away" after they are no longer needed. The arrows at the head of each generation portray the two possible fates for a log record near the head. If the record is garbage, it is ignored (conceptually thrown away in the garbage pail). If it is non-garbage, then it must be retained in the log and so it is either forwarded to the tail of the next generation or recirculated in the last generation. A stable version of the database resides elsewhere on disk. It does not necessarily incorporate the most recent changes to the database, but the log contains sufficient information to restore it to the most recent consistent state if a crash were to occur. The arrows underneath each generation illustrate the flushing activity that occurs in parallel with logging, and indicate that the log records whose updated objects are flushed may, in general, be anywhere in the log.

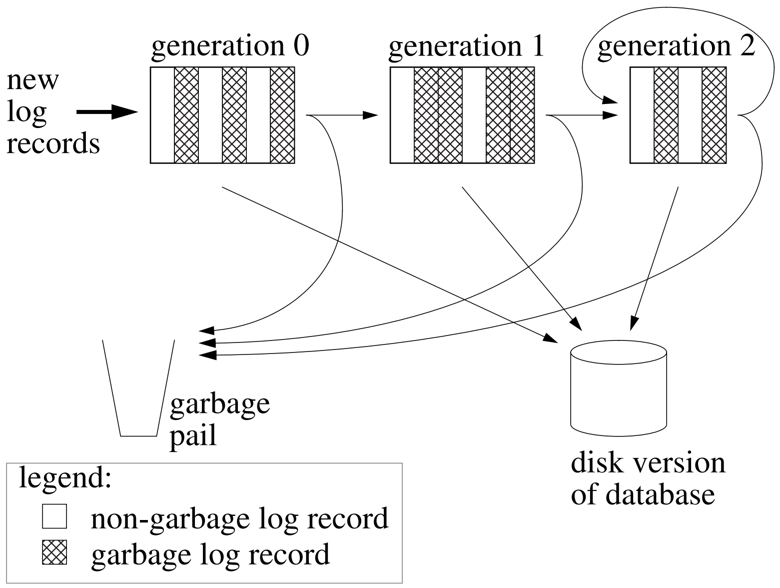


Figure 2.3: Disk Space Management Using XEL

This segmentation of the log is particularly effective if a large proportion of transactions finish execution and have their updates flushed before their log records near the head of generation 0. Many, if not all, of these records become garbage before the LM must decide whether or not to forward them and so the LM does not forward them to generation 1; their disk space can quickly be reclaimed for more incoming log records.

Only a small proportion of log records, mostly from transactions with longer lives, are forwarded to subsequent generations.

Recirculation in the last generation means that the physical order of its records no longer necessarily corresponds to the temporal order in which they were originally generated. The LM includes timestamps in data log records to enable the RM to establish the temporal order of the records.

2.3 Types and Statuses of Log Records

The previous section described the segmentation of a log stream into a chain of fixed-size FIFO queues and mentioned that the LM performs garbage collection at the head of each queue. This section will explain the basis upon which the LM decides whether or not a particular log record is garbage. There are several types of log records. For each type of log record, a record may be in any one of several possible states at any given time. In response to ongoing activity in the database, the state of a record may change over time. The LM's decision about whether to retain or throw away a log record depends on the record's state.

XEL performs physical state logging on the access path level, according to the taxonomy of [30]. In short, XEL performs redo logging with lazy logging of undo records. It adheres to the write ahead log (WAL) protocol [5]: the disk version of the database cannot be modified before the LM has written a log record to disk which describes the modification.

There are two types of log records. *Data log records (DLRs)* chronicle changes to the contents of the database (creation, modification or deletion of objects). *Transaction (tx) log records (TLRs)* mark important milestones (e.g., begin, commit or abort) during the lives of transactions.

Apropos TLRs, XEL logs only commit events; it does not bother to log even the commit of a transaction that did not update any objects in the database. When a transaction (that updated at least one object) successfully terminates, the LM adds a `COMMIT` record to the log to mark the occasion. The `COMMIT` record holds only the transaction's identifier. Previous logging and recovery methods also logged transactions' begin and abort events. XEL can incorporate these other types of TLRs, but they are superfluous. These anachronistic TLR types played an important role in previous recovery algorithms but are no longer relevant for XEL's recovery algorithm.

Figure 2.4 illustrates the noteworthy events in the lifetime of a typical transaction. The transaction begins, updates several objects and then requests to commit. Whenever the transaction updates an object, the LM writes a DLR to the log. The transaction can continue executing without needing to wait for the DLR to be written to disk. If the transaction eventually requests to commit, the LM generates a `COMMIT` record for it and writes the record to the log. After all the transaction's log records have been written to disk, the LM acknowledges the transaction's request to commit, thus bringing its course of execution to a close.

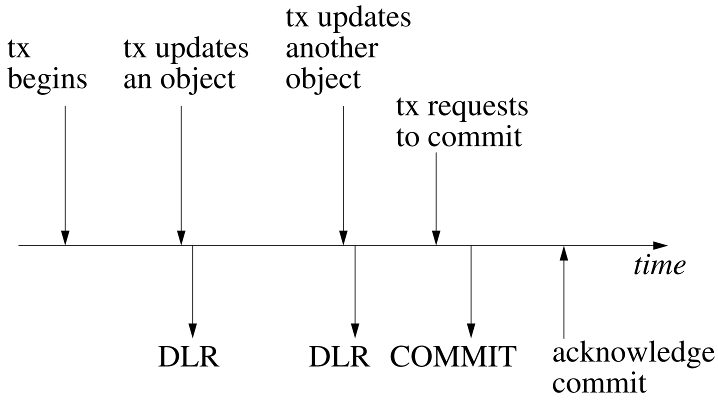


Figure 2.4: Typical Events During the Lifetime of a Transaction

There are two varieties of DLRs: *REDO* and *UNDO* DLRs. The LM generates a REDO DLR whenever a transaction modifies an object in the database. Each REDO DLR contains the following four pieces of information:

oid:	identifier for the affected object
txid:	identifier for the transaction that performed the update
timestamp:	indication of when the update occurred
new-value:	new value of the object

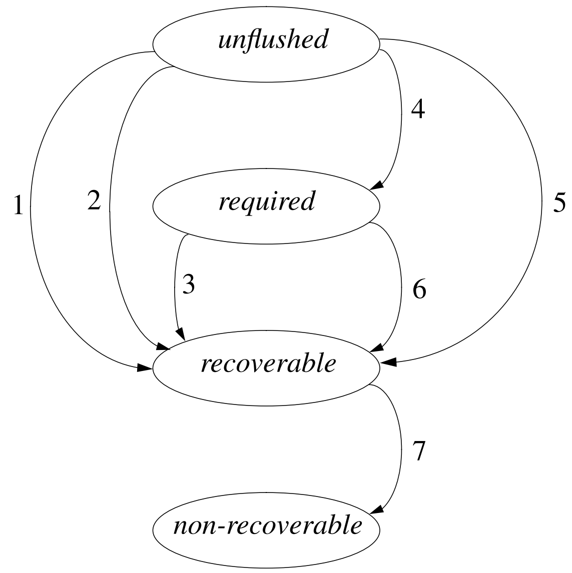
If the CM wants to flush an uncommitted update out to the disk version of the database, it must first inform the LM of its intentions and obtain permission from the LM. In response to such a request, the LM generates an UNDO DLR with the following pieces of information:

oid:	identifier for the affected object
txid:	identifier for the transaction that performed the update
timestamp:	indication of when the update occurred
old-value:	old value of the object (prior to start of transaction)

The LM grants permission to the CM to flush the uncommitted update only after the UNDO DLR has been written to disk. It is unnecessary for the LM to generate more than one UNDO DLR for a particular object and a particular transaction.

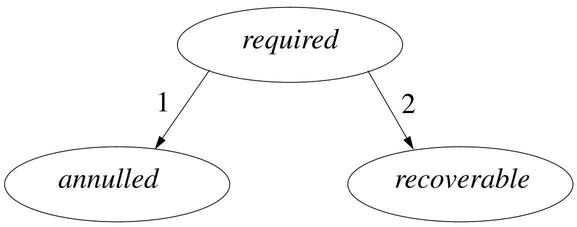
The LM must write a REDO DLR to the log for every update, but it is expected that only a very few updates, mostly from exceptionally lengthy transactions that modify a large number of objects, will trigger UNDO DLRs as well. Therefore, UNDO DLRs ought to be quite rare, in general.

Each log record must be in a particular state at any given time. The LM maintains data structures in main memory that track of the state of all log records; a record's state information is not kept in the log record itself. Figures 2.5, 2.6 and 2.7 graphically summarize the states and transitions for REDO DLRs, UNDO DLRs and `COMMIT` TLRs, respectively. Subsequent paragraphs will explain these state transition diagrams in detail.



Transition events:
1) Transaction updates same object again
2) Commit of more recent update to same object
3) Commit of more recent update to same object
4) Flush completed; older recoverable DLR still exists
5) Flush completed; no older recoverable DLR exists
6) Last older recoverable DLR becomes non-recoverable
7) Transaction's COMMIT record is overwritten

Figure 2.5: State Transition Diagram for a REDO DLR



Transition events:
1) Transaction commits
2) Aborted update undone by cache manager

Figure 2.6: State Transition Diagram for an UNDO DLR

A REDO DLR may have one of four different status values: *unflushed*, *required*, *recoverable* and *non-recoverable*. If a REDO DLR holds a more recent value for its asso-

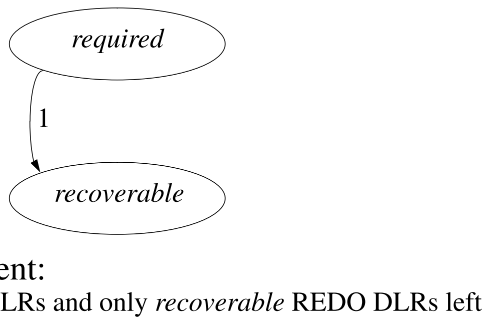


Figure 2.7: State Transition Diagram for a `COMMIT` TLR

ciated object than does the disk version of the database, the REDO DLR corresponds to the most recent modification to its associated object by the transaction that performed the update[2] and there is no REDO DLR in the log for a more recently committed update to the same object, then the DLR must have a status of *unflushed.* A *required* DLR must be retained in the log in order to ensure correct recovery after a crash. A REDO DLR with a status of *recoverable* can be recovered by the RM after a crash (because the `COMMIT` TLR from the corresponding transaction is also still in the log on disk), but is not required for correct recovery. A REDO DLR whose status is *non-recoverable* cannot be recovered after a crash because the `COMMIT` TLR from its transaction has already been overwritten on disk by more recent log records.

An UNDO DLR can have one of three status values: *required*, *annulled* and *recoverable.* An UNDO DLR initially has status *required* when the LM creates it. It remains *required* until the transaction that wrote it commits, at which time it becomes *annulled*; the UNDO DLR remains *annulled* until it is overwritten on disk. If a transaction writes an UNDO DLR and later aborts, the UNDO DLR retains its *required* status until the CM restores the corresponding object in the disk version of the database to the value that it held prior to the aborted transaction's update. Note that the CM need not immediately write out the object's original value to the disk version of the database; it can buffer the undo operation until a convenient opportunity, so as to achieve better disk I/O. After the CM has undone the aborted update (by restoring the object's representation in the disk version of the database to its original value), the LM changes the status of the corresponding UNDO DLR to *recoverable.* A *recoverable* UNDO DLR remains *recoverable* until it is eventually overwritten on disk.

There are two status values for a `COMMIT` TLR: *required* and *recoverable.* A `COMMIT` TLR has status *required* if at least one UNDO DLR (which may have a status of either *required* or *annulled*) that was written by the transaction still exists or if at least one REDO DLR that was written by the transaction has a status of *unflushed* or *required*; otherwise (i.e., no UNDO DLRs and any remaining REDO DLRs have *recoverable* status), the TLR has status *recoverable.*

[2] If a transaction modifies a particular object more than once, then the REDO DLRs for all updates except the most recent one have status *recoverable.*

Whenever an executing transaction updates an object, it writes a REDO DLR to the log. This DLR has *unflushed* status. The LM downgrades the status of any REDO DLRs from earlier updates to the same object by the same transaction to *recoverable*. If the transaction eventually commits, its `COMMIT` TLR has status *required*. Suppose that transaction t modifies an object x and commits. At the time that t commits, the LM checks if there is an *unflushed* or *required* REDO DLR for object x from an update by some earlier transaction. If such a DLR exists, the LM downgrades that DLR's status to *recoverable*.

The CM may flush an updated object to the disk version of the database whenever it chooses. In general, the CM attempts to flush all a transaction's updates after it has committed, so that no UNDO DLRs are needed; nevertheless, the CM may occasionally need to flush some of a transaction's updates to disk before the transaction commits. As soon as the CM has flushed an update to some object, the LM downgrades the status of any *unflushed* REDO DLR from an earlier transaction. The LM assigns a status of *required* to an earlier REDO DLR if it corresponds to the most recently committed update to the object and must be retained because of lingering *recoverable* DLRs from earlier updates to the same object; otherwise, it assigns a status of only *recoverable* to the earlier REDO DLR. After processing any *unflushed* previous DLR for the object, the LM then processes the REDO DLR for the update that was just flushed. If this DLR still has *unflushed* status at the time the flush operation completes[3] and there exists a *required* or *recoverable* DLR from an earlier update to the object, then the LM downgrades the DLR's status to *required*; otherwise, the LM assigns a status of only *recoverable* to the DLR whose update was just flushed. The LM downgrades a *required* REDO DLR's status to *recoverable* after there is no longer a *required* or *recoverable* (REDO or UNDO) DLR from any earlier update to the corresponding object.

The LM downgrades a transaction's `COMMIT` TLR to status *recoverable* as soon as there is no longer any *unflushed* or *required* REDO DLR nor any UNDO DLR remaining from the transaction. When a transaction's `COMMIT` TLR is eventually overwritten on disk, the LM changes the status to *non-recoverable* for any remaining REDO DLRs that the transaction wrote. Note that this may trigger status changes for REDO DLRs and TLRs from more recent transactions.

The following pseudocode expresses how the LM manages the states of records in the log.

```
create_new_record(log_record) {
   if (type of log_record is REDO DLR) {
         status of new record ← unflushed
         if (log_record's transaction previously updated same object) {
            status of REDO DLR from previous update ← recoverable
```

[3]A REDO DLR may have a status of *unflushed* at the time that the CM decides to initiate a flush operation for it, but the DLR may be rendered *recoverable* by a more recently committed update before the flush operation completes.

```
            }
        }
        else {
            status of new record ← required
        }
}


commit_transaction(txid) {
    for every object updated by transaction txid {
        if (unflushed or required REDO DLR remains from earlier transaction) {
            status of REDO DLR from earlier transaction ← recoverable
        }
        if (an UNDO DLR for the object was written out to the log) {
            status of UNDO DLR ← annulled
        }
    }
}


abort_transaction(txid) {
    for every object updated by transaction txid {
        for every update to the object {
            status of REDO DLR from the update ← non-recoverable
        }
        if (uncommitted update was flushed to disk version of database) {
            retrieve UNDO DLR from log
            request CM to restore object in disk version of database to original value
        }
    }
}


change_redo_to_recov(log_record) {
    status of log_record ← recoverable
    tid ← txid of transaction that write log_record
    if (            (no more unflushed, required or annulled DLRs from txid)
          AND   (txid has committed)) {
        status of COMMIT record from txid ← recoverable
    }
}


aborted_update_undone(object_id, txid) {
    status of UNDO DLR for update to object_id by txid ← recoverable
```

```
}

update_written_to_disk_version_of_db(object_id, txid, timestamp) {
   for every REDO DLR r from a previous update that has status unflushed {
      if (          (r is for most recently committed update to object_id)
           AND   (older recoverable DLRs for object_id still exist)) {
            status of r ← required
         }
         else {
            status of r ← recoverable
         }
   }
   if (status of DLR from flushed update is still unflushed) {
      if (recoverable DLRs from previous updates are in log) {
            status of REDO DLR for flushed update ← required
         }
         else {
            change_redo_to_recov(REDO DLR for flushed update)
         }
   }
}

record_erased(log_record) {
   case (type of log_record) {
      REDO DLR:
         if (status of oldest surviving REDO DLR for the object is required) {
            change_redo_to_recov(oldest surviving REDO DLR for object)
         }
      UNDO DLR:
         if (no unflushed, required or annulled DLRs from log_record's tx) {
            status of COMMIT record for log_record's transaction ← recoverable
         }
      COMMIT:
         for every remaining recoverable REDO DLR from log_record's tx {
            status of REDO DLR ← non-recoverable
            if (status of oldest surviving REDO DLR for the object is required) {
               change_redo_to_recov(oldest surviving REDO DLR for object)
            }
         }
   }
}
```

2.4 Management of Log Records

The previous section explained the basis upon which the LM decides whether or not to keep a log record. This section introduces the data structures which enable the LM to make this decision. The LM keeps a pointer to every noteworthy log record. A record's pointer indicates its location in the log and current status. The LM uses this information when it must decide a log record's fate.

It is convenient to broadly classify log records as *relevant* or *irrelevant*. All records that can affect the recovered state of the database are collectively referred to as *relevant* log records; *unflushed*, *required* and *recoverable* REDO DLRs and all UNDO DLRs are relevant log records, as are all *required* `COMMIT` TLRs and *recoverable* `COMMIT` TLRs for which some corresponding *recoverable* REDO DLRs still remain. All other log records (*non-recoverable* DLRs and every *recoverable* `COMMIT` record for which no corresponding REDO DLRs remain) are *irrelevant*. The LM must keep track of the positions of all relevant records. It does not bother to keep track of irrelevant records.

A *cell* exists for every relevant record in any generation of the log. Each cell resides in main memory and points to the record's location on disk. A record's location on disk, as pointed to by its cell, is indicated by an identifier of the block to which it belongs; finer granularity (e.g., position within the block) is not required by XEL. The cells corresponding to each generation are joined in a doubly linked list that "wraps around" in a circular manner; the cells nearest the head and tail have right and left pointers to each other, respectively. For generation i, pointer h_i points to the cell for the relevant record nearest the head. There is no tail pointer for a generation, but the cell for the relevant record nearest to the tail can be found quickly by following the right pointer of the cell pointed to by h_i.

The *logged object table (LOT)* has an entry for every object that has at least one relevant DLR somewhere in the log. An object's LOT entry keeps track of the positions within the log of its relevant DLRs. Cells for an object's relevant DLRs are accessible via its LOT entry.

Likewise, the *logged transaction table (LTT)* has an entry for every transaction that has updated at least one object. A transaction's LTT entry keeps track of all objects that it updated and for which the corresponding REDO DLRs are still relevant. After a transaction commits, its LTT entry points to the cell that corresponds to its `COMMIT` TLR.

The LM continually updates the LOT and LTT to reflect the current state of the system as transactions and log records come and go. At any given time, the cells associated with the LOT and LTT entries point to all relevant log records. Although cells belong to these two different tables, they may nonetheless simultaneously belong to the same doubly linked list.

An example of XEL with N=3 generations is shown in Figure 2.8. To relate Figure 2.8 to Figure 2.3, note that all irrelevant records are garbage records. Some relevant log records can affect recovery but are not required for recovery, and so they are also garbage records; they can be thrown away with impunity when convenient.

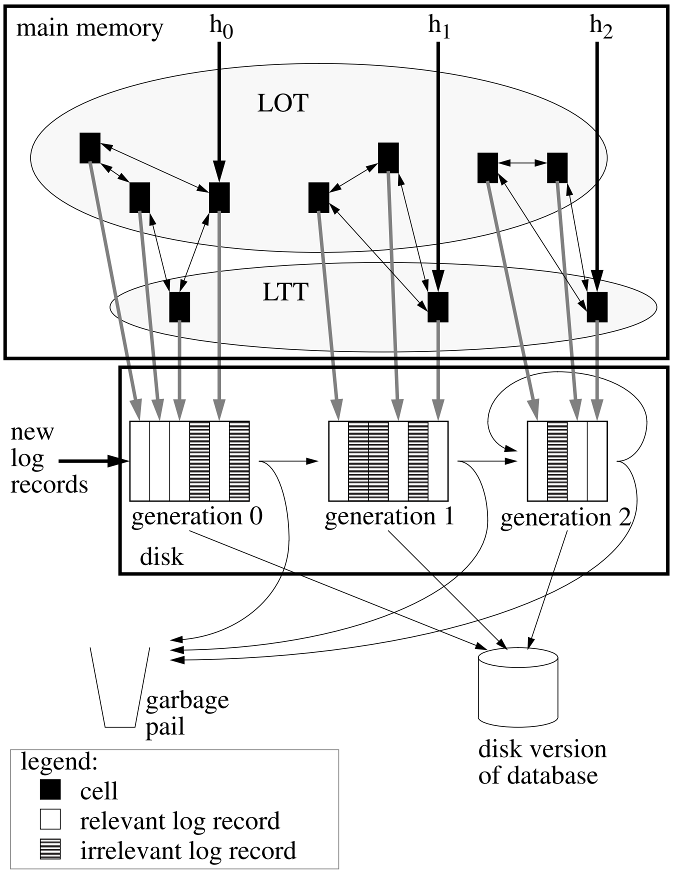


Figure 2.8: Data Structures for XEL

Figure 2.8 illustrates the most important aspects of XEL's data structures. The LOT and LTT, with their constituent cells, reside in main memory. Other internal details of the LOT and LTT have been omitted; the circular doubly linked lists of cells are the important aspect of the LOT and LTT in this figure.

Each cell has a status field that indicates the status of its corresponding log record. At any given time, the LM can determine whether a record is non-garbage by checking the status field in the record's cell. When a record must be forwarded to the tail of generation i+1, the LM writes its contents to disk at the tail of generation i+1, updates its cell, c, to point to its new position in the log and transfers c from the circular linked list for generation i to the circular linked list for generation i+1. The LM updates pointer h_i to point to the cell previously to the left of c, if such a cell exists for generation i; otherwise, it sets h_i to *NULL*. If h_{i+1} was *NULL* immediately before the record was forwarded, then the LM updates it to point to c (and c's left and right pointers point to itself). Recirculation in the last generation is handled similarly.

2.5 Buffer Management for Disk Version of the Database

The LM relies on the CM to flush updated objects to the disk version of the database so that log records from these updates will become garbage. This section briefly discusses how the CM ought to schedule flush operations and elaborates on how the CM interacts with the LM.

In general, there is negligible locality of access between the updates of independent transactions. Flushing updates in the order that they are written to the log would lead to random disk I/O for the disk version of the database. Instead, the CM maintains a pool of objects waiting to have their committed updates flushed and schedules writes to disk so that it can take advantage of locality in the disk version of the database and thus improve I/O performance. Ideally, there should usually be a significantly large number of committed updates from which the CM can choose the next object to be flushed; too small a "pool" of updates leads to random I/O. Flushing can proceed continuously at as high a rate as possible.

Occasionally, the CM may need to flush uncommitted updates out to the disk version of the database (because its buffer pool is running dangerously low on free space, for example). In such an emergency situation, the CM must first obtain permission from the LM, as described in section 2.3. After the LM has written the necessary UNDO DLRs to disk and granted the CM permission to flush some uncommitted updates to disk, the CM can schedule the writes to disk so as to exploit locality.

In the rare event that a transaction aborts after writing one or more UNDO DLRs, the CM must undo the transaction's updates that have already been propagated to the disk version of the database. The LM reads (from disk) every UNDO DLR that was written by the aborted transaction. For each such UNDO DLR, the LM communicates the oid and old value to the CM. In response, the CM restores the object (in the disk version of the database) to the value that it had prior to the aborted transaction's update and informs the LM after it has undone the transaction's modification to the object.

2.6 Flow Control

This section discusses how the LM regulates the flow of log records from one generation to the next in a log stream. Each generation is a FIFO queue of fixed size. If it begins to run out of space for new records, the LM must try to free up some space by throwing away or forwarding (or recirculating) log records from near the head of the queue. The

LM can forward records from one generation only if the next generation is able to accept them. Hence, flow control between successive generations within a log stream must be regulated.

The LM attempts to keep at least N_{free} blocks available in each generation to accept incoming (and recirculated, in the case of the oldest generation) log records. When the tail of generation i advances so as to violate this "low water mark", the LM attempts to forward (or recirculate) log records from generation i. However, the LM can forward log records to generation i+1 only if generation i+1 is able to accept them.

The LM refuses to overwrite any block that holds an *unflushed* or *required* log record. If h_{i+1} points to a record in the block immediately after the current tail position of generation i+1, then generation i+1 cannot accept any forwarded log records. Similarly, the LM refuses to accept any new log records from client transactions if space is not available for them in generation 0.

When space eventually becomes available in generation i+1, the LM will resume forwarding of records from generation i if there are fewer than N_{free} blocks between the current tail position of generation i and the block to which h_i indirectly points (h_i points to a cell, and this cell points to a block position on disk).

This flow control policy is guaranteed to be free of deadlock. The LM never needs to keep a log record because of the lingering presence of some chronologically subsequent log record. This property ensures that the dependency graph amongst log records is acyclic, and therefore deadlock is impossible.

In summary, a producer-consumer protocol between adjacent generations regulates the flow of forwarded log records. Older generations that become full exert "backpressure" on younger generations. If all generations become full, then the LM does not accept log records from client transactions. This policy ensures that necessary log information is never lost.

2.7 Buffering, Forwarding and Recirculation

The previous section explained how the LM regulated the flow of records into and out of each generation of a log stream. This section elaborates on some important timing details which govern this movement of log records. Much of the complexity arises from the characteristics and limitations of current disk drive technology.

Two characteristics of current disk technology exert an important influence on the implementation of XEL. Information is written to disk in fixed sized blocks (with each block typically some multiple of 1024 bytes). Sequential disk I/O is faster than random disk I/O. XEL must accommodate the constraint of fixed sized disk blocks, and ought to take advantage of the performance benefits of sequential I/O.

The LM uses the *group commit* technique [15, 5]. Records are collected in a buffer and written to disk all at once. Figure 2.9 illustrates the group commit technique. The bottom of the buffer holds log records that have already arrived. The LM adds new incoming log records to the buffer by putting them in the unfilled portion shown at the top of the buffer. In Figure 2.9, the direction of growth is upward.

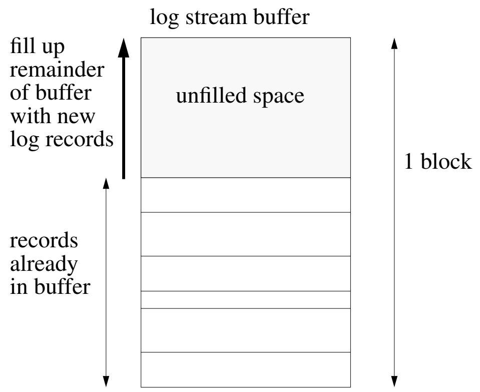


Figure 2.9: Buffering of Incoming Log Records for Batched Write to Disk

The LM should dedicate at least two buffers for log records that are to be written to generation 0 because a disk write generally requires a significant amount of time, such as 10 ms, during which other log records may arrive. While one buffer is being written to disk, the LM can add new records to a different buffer without risk of interference. The size of each buffer in the pool is exactly equal to the size of a disk block. At any given time, there is a current buffer for generation 0. The LM adds new log records to this buffer until it is full or a time limit runs out, at which time the LM writes it to disk; another buffer in the pool becomes the current buffer as soon as there are no *unflushed* or *required* records in the block to which the new current buffer will be written. Therefore, log records are not immediately written to disk. There is a delay while the current buffer fills, and some extra delay for the disk I/O.

Only one buffer is needed for each generation $i>0$ because the LM has the liberty of scheduling the movement of log records between generations. There can be only one outstanding write to the tail of a particular generation and the LM can quickly refill generation i's buffer as soon as the current write operation completes, so additional buffers would not help anyway.

The tail of generation i points to the location of a block on disk; finer granularity is unnecessary. When a new log record comes in to generation i, the LM will attempt to allocate it to the block indicated by the current position of the tail; if there is insufficient room remaining in the current buffer to accommodate the record, then the LM will attempt to advance the tail block position and allocate the record to the new tail block.

The head of generation i is the block to which h_i (indirectly) points, if h_i is not *NULL*. If h_i is *NULL*, then the head of generation i is, by default, the current tail block.

The movement of head and tail pointers in block sized quanta has implications. When the LM decides to advance the head of generation i, it must deal with all log records in the head block. The LM attempts to forward all *unflushed* and *required* records in this head block to generation i+1. Suppose that the LM can forward these records to generation i+1. It adds the records to the buffer for generation i+1. In general, they are insufficient to completely fill the buffer, but the LM must ensure that the forwarded records are soon written to disk in generation i+1. Therefore, it attempts to fill the buffer as full as possible before writing it. After forwarding records from the block at the head of generation i, the LM works backward (i.e., to the left) from the head to gather enough other non-garbage log records to fill the buffer that is destined for the tail of generation i+1. In summary, the requirements of generation i dictate that records be removed from its head in quanta of size at least a block. The requirements associated with forwarding records to the tail of generation i+1 imply that records are usually forwarded as a group from the first several blocks at the head of generation i.

There are two details that complicate the operation of forwarding records from generation i to generation i+1. First, there is some delay between the time when the LM decides to forward a log record to the moment when the forwarded record is actually on disk in generation i+1. Second, two copies of a forwarded record may temporarily exist in the log. The forwarded copy of the record resides on disk in generation i+1, but there is also the "stale" copy left behind in generation i; this latter copy remains in the log until it is overwritten by newer log records. Because of these details, the LM manipulates two additional special pointers for each generation.

For each generation i, s_i is the *scan pointer*. Like h_i, it points to a cell in the circular doubly linked list for generation i or is *NULL*. It indicates how far the LM has scanned to forward log records to generation i+1. If there is no forwarding operation in progress, then s_i coincides with h_i. When the LM examines a log record and decides to forward it, it leaves the cell in generation i's list and advances s_i to the left; if this leftward movement causes s_i to "wrap around" so that it comes back to h_i, then the LM sets it to *NULL* instead. Immediately after a buffer of forwarded records has been written to disk in generation i+1, the LM keeps advancing h_i leftward until it coincides with s_i or becomes *NULL*; until h_i "catches up" to s_i, the LM transfers each cell over which it passes from generation i's circular list of cells to the tail of generation i+1's list. This cautious management of the h_i and s_i pointers ensures that the LM never inadvertently overwrites a non-garbage log record in generation i before it has been forwarded to generation i+1 (and is on disk in generation i+1).

The LM maintains a second circular doubly linked list of cells for every generation. This other list is called the *doomed* list because it indicates relevant records that will (soon) be overwritten. The pointer d_i points to the cell at the head of generation i's doomed list. When the LM examines a log record's cell and decides that the record is garbage, it removes the cell from the regular list (after advancing s_i, of course, and possibly also h_i if necessary) and adds it to the tail of the doomed list. If d_i was previously *NULL*, it will point to the cell that has just been transferred to the doomed list; otherwise, the transferred cell is the target of the right pointer of the cell to which d_i points. Now consider the case that the LM decides the log record to which s_i points is non-garbage. It creates a new cell that also points to the record and leaves the old cell intact in the regular list for generation i (where it waits to be forwarded to generation $i+1$ after the buffer has been written to disk in that generation). If the log record is a DLR, the LM adds the new cell to the LOT entry of the associated object; otherwise, it adds the new cell to the LTT entry of the corresponding transaction. Finally, the LM inserts this new cell into the doomed list at its tail. Whenever a block of log records is written to disk in generation i, the LM examines d_i. If d_i (indirectly) points to the block that has just been written, then the LM concludes that the record associated with the cell to which d_i points is now gone and so it moves d_i leftward (or assigns d_i the value *NULL*, if appropriate) and deletes the cell to which d_i had pointed. The LM continues to advance d_i leftward until it becomes *NULL* or no longer (indirectly) points to the block that has just been written. The d_i pointer ensures that the LM does not forget about any stale copy of a relevant log record.

Recirculation is not as complicated. The LM recirculates records from only the block at the head of the last generation and places them in a buffer without immediately writing it to disk. The existing copies of these records will not be overwritten before the tail has advanced, but the recirculated copies will belong to the disk block written at the tail. There is no need for a scan pointer in the last generation; the LM immediately transfers the cells for the recirculated records from head to tail in the circular list (in practice, this is accomplished simply by advancing h_i to the left). Similar to the case of forwarding, the LM transfers the cells for garbage log records to the doomed list and eventually deletes the cells after the log records have actually been overwritten. As new log records come in to the last generation, the LM adds them to the buffer after the recirculated records.

If the LM ever decides to delete the cell to which h_i points, it must first adjust h_i accordingly. If there are other cells in generation i's linked list, then the LM advances h_i to the left; otherwise, it assigns *NULL* to h_i. The LM behaves similarly if it deletes the cell to which s_i or d_i points.

In summary, the LM collects records in a buffer before writing them to any generation. It attempts to fill a buffer as full as possible before writing it to disk. When the LM decides to forward a log record, it does not transfer the record's cell from the circular list of generation i to the list of generation $i+1$ until after it is certain that the record is on disk in generation $i+1$. The LM keeps track of the positions of all copies of all relevant log records until they are actually overwritten on disk (or until they become irrelevant, if this happens first).

The following pseudocode routines succinctly state the LM's algorithms for forwarding and recirculating log records.

```
add_cell_to_generation(cell, i) {
    if (h_i==NULL) {
            h_i ← cell
            s_i ← cell
            cell−>left ← cell
            cell−>right ← cell
        }
        else {
            if (s_i==NULL) {
                s_i ← cell
            }
            cell−>left ← h_i
            cell−>right ← h_i−>right
            cell−>left−>right ← cell
            cell−>right−>left ← cell
        }
}


delete_cell_from_generation(cell, i) {
    if (s_i==cell) {
        if (s_i−>left==h_i) {
                s_i ← NULL
            }
            else {
                s_i ← s_i−>left
            }
    }
    if (h_i==cell) {
        if (h_i−>left==h_i) {
                h_i ← NULL
            }
            else {
                h_i ← h_i−>left
            }
    }
    if (d_i==cell) {
        if (d_i−>left==d_i) {
                d_i ← NULL
            }
            else {
                d_i ← d_i−>left
            }
```

```
    }
    if (cell−>left≠cell) {
        cell−>left−>right ← cell−>right
        cell−>right−>left ← cell−>left
    }
}


add_cell_to_doomed_list(cell, i) {
    if (d_i==NULL) {
            d_i ← cell
            cell−>left ← cell
            cell−>right ← cell
        }
        else {
            cell−>left ← d_i
            cell−>right ← d_i−>right
            cell−>left−>right ← cell
            cell−>right−>left ← cell
        }
}


forward_records_from_generation(i) {
    while ((generation i+1 can accept records) AND (s_i≠NULL)) {
        if (record pointed to by s_i must be kept) {
                copy record pointed to via s_i to buffer for generation i+1
                create new_cell
                copy contents of cell pointed to by s_i into new_cell
                add_cell_to_doomed_list(new_cell, i)
                if (s_i−>left==h_i) {
                        s_i ← NULL
                    }
                    else {
                        s_i ← s_i−>left
                    }
            }
            else {
                cell ← s_i
                delete_cell_from_generation(cell, i)
                add_cell_to_doomed_list(cell, i)
            }
    }
}
```

```
recirculate_records_within_generation(i) {
   while (fewer than N_free blocks available in generation i) {
      if (record pointed to by h_i must be kept) {
            copy record pointed to via h_i to buffer for generation i
            h_i ← h_i−>left
         }
         else {
            cell ← h_i
            delete_cell_from_generation(cell, i)
            add_cell_to_doomed_list(cell, i)
         }
   }
}


buffer_written_to_generation(i) {
   if (i>1) {
      while ((h_{i−1}≠NULL) AND (h_{i−1}≠s_{i−1})) {
         cell ← h_{i−1}
         delete_cell_from_generation(cell, i-1)
         add_cell_to_generation(cell, i)
      }
   }
   while ((d_i≠NULL) AND (d_i points to block position just overwritten)) {
      record_erased(d_i)
      if (d_i−>left==d_i) {
            d_i ← NULL
         }
         else {
            d_i ← d_i−>left
         }
   }
}
```

2.8 Management of the LOT and LTT

Section 2.4 introduced the LOT and LTT when it described the doubly linked lists of cells that track the positions of all relevant log records in the generations of a log stream. This section provides more details about the LOT and LTT and describes how the LM manages these data structures.

The LOT and LTT keep track of all relevant log records. The LM updates them on a continual basis as records enter the log and progress through it.

The LM associatively accesses each object's LOT entry by using its object identifier (*oid*) as a key. A hash table implementation is therefore appropriate. The dynamic nature of the LOT strongly suggests that *chaining* [10] (rather than *open addressing*) is the most suitable technique for collision resolution. An object's LOT entry has one or more cells, each of which points to the disk block of a relevant DLR for the object. The LM manages these cells as a linked list.

Entries in the LTT are associatively accessed using transaction identifiers (*tids*) as keys. Like the LOT, the LTT is implemented as a hash table with chaining for collision resolution. Each transaction's LTT entry holds a set *obj_ids* of oids to keep track of which objects were updated by the transaction; this set is initially empty and grows as the transaction progresses and performs work.

The LM maintains a timestamp in each object's LOT entry, although no timestamp is necessarily stored with any object in the disk version of the database. A simple integer-valued counter suffices for the timestamp. When the LM creates a new LOT entry for an object, it initializes the timestamp to 0. Whenever a transaction updates the object, the LM increments the timestamp and then puts the new timestamp value in the resulting REDO DLR. An UNDO DLR for an object holds the current value of the timestamp in its LOT entry at the time that the UNDO DLR is created; the LM does not bother to increment the timestamp when it creates an UNDO DLR. The LM removes an object's LOT entry only after it has no more relevant DLRs remaining in the log (the LM detects this situation when the set of cells associated with the LOT entry becomes empty). Therefore, at any given time, all relevant REDO DLRs for an object have unique timestamps and these DLRs can be placed into chronological sequence by their timestamps. Likewise, all UNDO DLRs have timestamps that indicate their chronological ordering. The cell for each DLR has a *tstamp* field that stores the value of the timestamp contained in the DLR.

Whenever a transaction modifies an object in the database, it causes the LM to send a REDO DLR to the log. If an entry does not already exist for the object in the LOT, the LM creates one. The LM increments the timestamp in the object's LOT entry, formats the DLR, adds the DLR to the current buffer for the tail of generation 0, creates a cell to point to the DLR's position in the log, adds it to the set of cells maintained in the object's LOT entry, inserts the cell in the doubly linked list for generation 0, creates a new LTT entry for the transaction that performed the update if it did not already have one and then adds the object's oid to the *obj_ids* set in the transaction's LTT entry.

Every transaction eventually commits or aborts. An abort is easy to handle. Because the log will never hold a `COMMIT` TLR from an aborted transaction, all the REDO DLRs from the transaction immediately become *non-recoverable*; the LM disposes the cells that pointed to these DLRs and deletes the transaction's LTT entry. However, any UNDO

DLRs from the transaction retain their *required* status after it aborts until the CM has undone these updates to the disk version of the database, as described in Sections 2.3 and 2.5.

When a transaction (which updated at least one object) commits, the LM updates its LTT entry so that it points to the cell for its `COMMIT` record in the log. Then the LM processes the members of *obj_ids* in the transaction's LTT entry. For each oid in *obj_ids*, the LM retrieves the object's LOT entry, assigns a status of *recoverable* to any *unflushed* or *required* REDO DLR from an earlier committed update to the same object, assigns a status of *annulled* to any UNDO DLR from the transaction that just committed and informs the CM that the most recent update has now been committed. If the CM has not already flushed this most recent update to disk, then the CM enqueues it to be flushed[4].

The LTT entry for each transaction includes a counter to keep track of the number of UNDO DLRs and *unflushed* or *required* REDO DLRs that exist for the transaction. The LM initializes an LTT entry's counter to 0 and increments this counter every time the transaction writes another REDO or UNDO DLR to the log. The LM also increments this counter whenever it copies an existing UNDO DLR so that it can forward the DLR. The LM decrements a transaction's counter each time that it downgrades the status of one of the transaction's REDO DLRs from *unflushed* or *required* to *recoverable*, or each time that it overwrites an UNDO DLR from the transaction. When the counter reaches zero, the LM downgrades the transaction's `COMMIT` TLR to *recoverable*.

After a transaction commits, its *obj_ids* set can only shrink in size. Whenever the last copy of a relevant REDO DLR is overwritten and no other REDO DLRs from other updates to the same object by the same transaction remain, the LM removes the corresponding oid from the *obj_ids* set of the transaction that wrote the DLR.

When the last copy of a transaction's `COMMIT` TLR is eventually overwritten, the LM examines its *obj_ids* set; for every object still represented in this set, the LM downgrades the status of all corresponding REDO DLRs to *non-recoverable* (and deletes the cells that pointed to these DLRs). Finally, the LM deletes the transaction's LTT entry.

If the *obj_ids* set in a committed transaction's LTT entry becomes empty and no UNDO DLRs remain from the transaction (as indicated by a counter value of zero), all copies of the transaction's `COMMIT` record become irrelevant. The LM disposes the cells that point to them and removes the transaction's entry from the LTT.

To summarize, every object with relevant DLRs in the log has an entry in the LOT. An object's LOT entry keeps track of the positions within the log of its relevant DLRs. There is an LTT entry for every transaction currently in progress that has updated

[4]If the CM already enqueued the object in response to an earlier update by another transaction but it has not yet flushed the object, then the object's oid remains unchanged in the set of objects waiting to be flushed.

at least one object and every committed transaction that still has relevant DLRs. A transaction's LTT entry keeps track of all objects that it updated and the positions within the log of copies of its `COMMIT` record. The LM continually updates the LOT and LTT to reflect the current state of the system as transactions and log records come and go. At any given time, the cells associated with the LOT and LTT entries point to all relevant records in the log.

2.9 Crash Recovery

After a crash has happened, the database invokes the RM to restore the disk version of the database to a consistent state. The RM examines the records in the log and attempts to find the most recently committed value, if any, for each object that has one or more DLRs in the log. The RM propagates each object's most recently committed value, if any, to the disk version of the database, thus restoring the disk version of the database to a consistent state: it incorporates all the effects of transactions which committed prior to the crash and none of the effects of transactions which aborted or were interrupted.

The RM starts sequentially reading from the disk(s) where the log is stored. It does not need to begin at the tail of generation 0 (nor of any other generation). The RM processes the log in a single pass. This new recovery algorithm is suitable for systems in which the log is not larger than main memory.

As each block is read from the log, the RM processes the DLRs and TLRs in the block. The *Pending Object Table (POT)* keeps track of all objects during the recovery process, while the *Recovered Transaction Table (RTT)* serves a similar purpose for transactions.

Each RTT entry belongs to a particular transaction. It holds two pieces of information about the transaction. The first is the *status* of the transaction. If the RM has found a `COMMIT` record for the transaction, it has a status of *committed*; otherwise, it has an *unknown* status. The RTT entry also contains a set of oids, called *pending_objs*. The contents of this set are meaningful only if the transaction's status is still *unknown*. Each member of *pending_objs* indicates an object for which a REDO DLR was already found in the log and this DLR was written by the transaction.

When the RM reads a REDO DLR from the log, it checks the POT entry for the associated object to see if a more recently committed REDO DLR or a more recent UNDO DLR has already been found (the timestamps within an object's REDO and UNDO DLRs indicate their relative temporal ordering). If not, it adds the new DLR to the POT. It also inspects the RTT to find out if the transaction that wrote the DLR is known to have committed. If the transaction did indeed commit, then the RM marks the new update as committed and deletes from the POT all earlier DLRs for the same object; otherwise, it leaves the update with a status of pending and adds the object's oid to the *pending_objs* set in the RTT entry of the corresponding transaction.

If the RM reads an UNDO DLR from the log, it ignores it if a more recently committed REDO DLR or a more recent UNDO DLR has already been found for the associated object; otherwise, it adds the UNDO DLR to the object's POT entry and deletes all DLRs that were written by earlier transactions (as indicated by their txid and timestamp fields).

When the RM upgrades a transaction's status from *unknown* to *committed* (in response to the discovery of a `COMMIT` TLR), it processes each object represented in the *pending_objs* set kept with the transaction's RTT entry. For each such object, it marks the corresponding update in the POT (if it still exists) as committed and deletes all earlier updates to the object; it also deletes the object's oid from the *pending_objs* set.

After all log records have been processed, the RM restores the disk version of the database to the most recent consistent state that existed prior to the crash by examining each object's POT entry and taking appropriate action. If an object's POT entry holds an UNDO DLR and the transaction that wrote the record has a status of *committed*, then the RM does nothing further for the object. However, an UNDO DLR from an uncommitted transaction[5] prompts the RM to propagate the object's value (indicated in the UNDO DLR) to the disk version of the database and thus undo the effect of the unsuccessful transaction. If an object's POT entry has a REDO DLR from a committed transaction, the RM flushes this updated value to the disk version of the database. The RM ignores any object whose POT entry holds neither an UNDO DLR from an uncommitted transaction nor a committed REDO DLR.

The following pseudocode expresses the RM's algorithms.

```
should_keep_redo_dlr(redo_dlr) {
   pot_entry ← POT entry for object redo_dlr−>oid
   redo_ts ← redo_dlr−>timestamp
   if (pot_entry has a committed REDO DLR with timestamp > redo_ts) {
         return FALSE
      }
      else {
         if (pot_entry has an UNDO DLR with timestamp > redo_ts) {
               return FALSE
            }
            else {
               if (pot_entry has a REDO DLR with timestamp == redo_ts) {
```

[5]The RM concludes that any transaction whose status is still *unknown* did not commit before the crash.

```
                return FALSE
            }
            else {
                return TRUE
            }
        }
    }
}


known_to_have_committed(txid) {
    if (RTT entry of txid has committed status) {
            return TRUE
        }
        else {
            return FALSE
        }
}


recover_redo_dlr(redo_dlr) {
    if (object redo_dlr−>oid has no POT entry) {
        create new POT entry for object redo_dlr−>oid
    }
    if (should_keep_redo_dlr(redo_dlr)) {
        add redo_dlr to POT entry of redo_dlr−>oid
        if (known_to_have_committed(redo_dlr−>txid)) {
                redo_dlr−>txid ← tx_committed
                delete all DLRs with timestamps less than redo_dlr−>timestamp
                delete any UNDO DLR with timestamp equal redo_dlr−>timestamp
            }
            else {
                add_to_pending_objs_in_rtt(redo_dlr−>oid, redo_dlr−>txid)
            }
    }
}


should_keep_undo_dlr(undo_dlr) {
    pot_entry ← POT entry for undo_dlr−>oid
    undo_ts ← undo_dlr−>timestamp
    if (pot_entry has a committed REDO DLR with timestamp ≥ undo_ts) {
            return FALSE
        }
        else {
```

```
        if (pot_entry has an UNDO DLR with timestamp ≥ undo_ts) {
            return FALSE
          }
          else {
            return TRUE
          }
      }
}


recover_undo_dlr(undo_dlr) {
   if (object undo_dlr−>oid has no POT entry) {
      create new POT entry for object undo_dlr−>oid
   }
   if (should_keep_undo_dlr(undo_dlr)) {
      add undo_dlr to POT entry of undo_dlr−>oid
      delete all other DLRs with timestamps less than undo_dlr−>timestamp
   }
}


update_pot_after_tx_commit(oid, txid) {
   if (POT entry for oid still has a REDO DLR from transaction txid) {
      redo_dlr ← REDO DLR for oid and txid with greatest timestamp
      redo_dlr−>txid ← tx_committed
      delete all other DLRs with timestamps less than redo_dlr−>timestamp
      delete any UNDO DLR with the same timestamp as redo_dlr−>timestamp
   }
}


recover_commit(commit_record) {
   if (commit_record−>txid has no RTT entry) {
      create new RTT entry for commit_record−>txid
   }
   status of transaction commit_record−>txid ← committed
   if (pending_objs≠∅ in RTT entry of commit_record−>txid) {
      for every oid in pending_objs {
         update_pot_after_tx_commit(oid, commit_record−>txid)
         remove oid from pending_objs
      }
   }
}
```

```
perform_recovery() {
   while (there are still some unread log records) {
      log_record ← read another record from the log
      case (type of log_record) {
           REDO DLR:    recover_redo_dlr(log_record)
           UNDO DLR:    recover_undo_dlr(log_record)
           COMMIT:      recover_commit(log_record)
      }
   }
   for every object with an entry in the POT {
      if (object's POT entry has an UNDO DLR from an uncommitted tx) {
            write out value from UNDO DLR to object in disk version of DB
         }
         else {
            if (object's POT entry has a committed REDO DLR) {
               write out value from committed REDO DLR to disk version of DB
            }
         }
   }
}
```

The LM ensures that the log always contains sufficient information for the RM to restore the database to a consistent state if a crash were to ever occur at any time. Consider a series of updates, performed by different transactions, to a particular object. Suppose that the most recent update to the object has been committed. If the CM has already flushed this update to the disk version of the database, then either no *recoverable* (UNDO or REDO) DLRs from prior updates to the object remain in the log or the DLR from the most recent update is still in the log (and has a status of *required*) along with the `COMMIT` record for the transaction which performed this update. If the CM has not already flushed this update, then its DLR must have a status of *unflushed* and is still in the log.

Now suppose that the most recent update to a particular object was performed by a transaction that aborted or is still in progress. Therefore, the log does not contain any `COMMIT` record from this transaction. If the log still holds a REDO DLR from this update, then it is innocuous anyway (because the RM ultimately ignores any REDO DLR from a transaction that did not commit). If the disk version of the database already holds the new uncommitted value for the object, then the log must hold an UNDO DLR which records the object's original value (i.e., the value which it had immediately prior to the beginning of the current transaction). Since the RM finds this UNDO DLR but it does not find a `COMMIT` record for the associated transaction, it restores the object in the disk version of the database to the value indicated in the UNDO DLR, thus undoing the uncommitted transaction's update. If the disk version of the database still holds the object's original value but the log holds an UNDO DLR for the uncommitted update (because the RM had requested the LM for permission to flush but had not yet performed the flush), then correct recovery is still ensured. Finally, if the log contains no UNDO DLR for the uncommitted update, then the disk version of the database must hold the object's original value and either the log holds a REDO DLR (with *unflushed*, *required* or *recoverable* status) from the most recently committed update or the log holds no *unflushed*, *required* or *recoverable* DLRs from prior updates to the object. Either way, the disk version of the database will hold the original value of the object after the RM finishes its work.

Therefore, the LM and RM together guarantee that the most recently committed value for every object is restored to the database after a crash, and thus the consistency of the database is maintained.

Note that the LOT and LTT data structures which played a crucial role during normal logging operations are unnecessary for recovery. They enabled the LM to manage the log's records so that the database could always be restored to a consistent state if a crash were to ever occur. After a crash has actually happened, the RM must examine the information which the LM left on disk and use it to restore the disk version of the database to a consistent state. When the RM has finished its work, the database resumes normal processing. The LM initializes the LOT and LTT data structures (they are initially empty) and then begins accepting requests from client transactions.

Chapter 3

Correctness Proof for XEL

This chapter presents a theoretical model for a simplified version of XEL and proves important safety and liveness properties. Effectively, the model ignores the role of the cache manager and assumes that every update is flushed to the disk version of the database immediately after it is committed. Nevertheless, the log manager must ensure that the REDO DLR from the most recently committed update to each object is retained until all prior REDO DLRs for the same object have first been rendered non-recoverable (review Section 2.1 to understand this requirement).

A manual correctness proof for a complete implementation of XEL, as presented in Chapter 2, would be overwhelming in terms of both size and effort; the proof itself would be prone to human error and its length would deter most readers from bothering to verify it. Nevertheless, this chapter does prove the correctness of a simplified version of XEL. The proof focuses on only a single object, but it applies to all objects in the database. Therefore, this simplified version of XEL ensures that the log always holds sufficient information for the RM to restore the database to a consistent state after a crash. Section 2.9 explained how the RM actually does restore the database to a consistent state, given the information in the log. This chapter also proves that every committed update's REDO DLR is eventually erased (a liveness property) so that its disk space can be reused.

Although the proof considers only a simplified version of XEL, it has worth nonetheless. After someone understands this proof for simplified XEL, they can extend this understanding so that it applies to more realistic implementations of XEL. This approach of starting reasonably simple and then gradually adding in more detail has pedagogic value. Furthermore, the experience of proving the correctness for a simplied version of XEL can suggest approaches for automating the many "mechanical" parts of the proof effort so that much more sophisticated implementations of XEL can be proven automatically by computer (assuming that the program which assists in the proof process is itself correct); little human effort would be required and so the chance of human error would be significantly reduced.

This chapter uses I/O automata theory [42, 43] extensively. A reader unfamiliar with I/O automata theory is referred to [42, 43] for an explanation of it.

The remainder of the chapter is organized as follows. Section 3.1 states what aspects of XEL are simplified and explains the interface for this much-simplified version of the log manager. It also suggests how to gradually embellish this simplified model so that a more realistic version of XEL is obtained. To perform correctly, any variation of XEL can reasonably require client transactions to behave appropriately; Section 3.2 formally expresses these restrictions as four well-formedness properties that the log manager's surrounding environment must always satisfy. Section 3.3 presents a model of the log manager that is as simple as possible and proves that it is correct, as long as the environment satisfies the well-formedness properties; this very simple model for the log manager is referred to as `SLM`. Section 3.4 defines a more complex model for XEL that more closely resembles a real implementation and then proves safety and liveness properties for it. The I/O automata description for this implementation is presented in Section 3.4.1. This implementation is referred to as `LM`. Even though `LM` is fairly elaborate, it still has many simplifications and does not constitute an implementation of the complete XEL technique as presented in Chapter 2. Section 3.4.3 postulates a *possibilities mapping* f that maps each state in `LM` to a set of states in `SLM` and then proves that f is indeed a possibilities mapping, according to the definition in [42, 43]. This result inductively proves that `LM` is correct apropos safety. For any possible execution of `LM`, a corresponding execution of `SLM` also exists which has exactly the same external behavior. Since the correctness (in terms of all possible external behaviors) of `SLM` has already been proven, the correctness of `LM` follows as a result. Finally, Section 3.4.4 states an important liveness property: every log record is eventually erased. Appendix A provides the many lemmas and theorems which constitute the safety and liveness proofs.

3.1 Simplifications

This section describes the log manager's interface and explains how the log manager interacts with the world around it. Subsequent sections will build upon the introductory description which this section provides. Because this chapter considers a simplified version of XEL for the log manager, the interface is simpler than what was described in Chapter 2. This section explains and justifies these simplifications. It also discusses how some of these simplifications would be relaxed if one wanted to extend the techniques of this chapter to a more realistic version of XEL.

This chapter will prove that XEL (in its simplified manifestation) does "the right thing" for each object. Each object is characterized by a set of possible updates. These updates may be sequenced in any particular order; the "external world" chooses the order by issuing a series of commit commands. Figure 3.1 illustrates the log manager's interface for the simplified version of XEL that is considered in this chapter. In this simple model, the log manager manages only a single object. The external world sends $\texttt{COMMIT}_i$ and $\texttt{ERASE}_i$ messages to the log manager, where the subscript i (which is a member of some index set) identifies a particular update, and the log manager sends $\texttt{ERASABLE}_i$ messages to the external world. Here, the external world is everything outside the log manager.

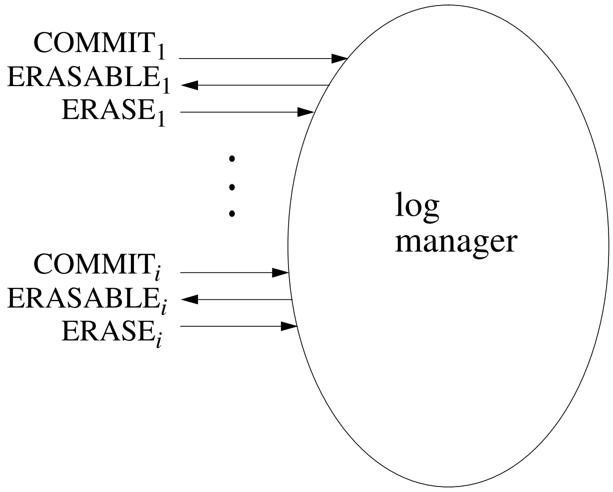


Figure 3.1: Interface of Simplified Log Manager

When the external world (specifically, some client transaction) wants to atomically update an object, it sends a $\texttt{COMMIT}_i$ request to the log manager; the subscript identifies the particular update which the external world wants to commit. The log manager keeps some (internal) record of this update but may eventually decide to delete it. When it decides that it no longer needs to retain a record from update i, it sends an $\texttt{ERASABLE}_i$ message to the external world. In response to this message, the outside world chooses exactly when the record from update i is actually deleted; the external world sends an $\texttt{ERASE}_i$ message to the log manager to inform it when the record from update i has been deleted. The $\texttt{ERASABLE}_i$ and $\texttt{ERASE}_i$ messages model the operation of a typical disk drive. When the log manager decides to overwrite a particular log record on disk, it submits a request to the disk drive's controller to write a block of new information to the record's location on disk; this request corresponds to the $\texttt{ERASABLE}_i$ message. After the disk drive has actually completed the write operation, it informs the log manager; this acknowledgement corresponds to the $\texttt{ERASE}_i$ message.

The log manager must satisfy the following important property. Either the log manager still retains the record from the most recently committed update to the object (i.e., $\texttt{COMMIT}_i$ has happened, no subsequent $\texttt{COMMIT}_j$ has occurred yet, and the log manager has not issued an $\texttt{ERASABLE}_i$ message) or the records from all previously committed updates have already been deleted (i.e., for every $\texttt{COMMIT}_h$ which preceded $\texttt{COMMIT}_i$, a corresponding $\texttt{ERASE}_h$ has already happened). This guarantees that no stale old update can jeopardize the state of the database if a crash were to occur.

The particular values of the updates to an object are irrelevant, so this simplified model makes no mention of them. It may help to think of update i's value being communicated in the $\texttt{COMMIT}_i$ message. A more elaborate model of XEL would choose to include a $\texttt{MODIFY}_i$ action by which the external world assigns a value to the object for update i. If the external world later decides to make this modification permanent, it submits a separate $\texttt{COMMIT}_i$ request; alternatively, an $\texttt{ABORT}_i$ message would annul the update.

The model ignores the role of the cache manager. Effectively, it assumes that each $\texttt{COMMIT}_i$ action simultaneously commits update i and flushes the object's new value (which was assigned by update i) to the disk version of the database. The log manager can grant permission to erase the REDO DLR from some update i as soon as the records from all chronologically preceding updates have been erased; it need not wait for the completion of a flush to the disk version of the database. Furthermore, UNDO DLRs do not play a role in this simplified model because a new value is (conceptually) assigned and committed at the same time. A more realistic model would include some additional $\texttt{FLUSH}_i$ input action to the log manager to inform it that the value which update i assigned to the object has been flushed to the disk version of the database; as long as update i is the most recently committed update, the log manager cannot issue an $\texttt{ERASABLE}_i$ message until $\texttt{FLUSH}_i$ has happened.

This model applies to only a single object. The fact that a transaction can update an arbitrary number of other objects is irrelevant. A more realistic model would embed the above model within a larger model that provides transactional support. In this larger model, a transaction would send $\texttt{MODIFY}_i$ messages to any number of objects. If the transaction later committed, the log manager would send a $\texttt{COMMIT}_i$ message to each object which the transaction updated. Hence, the above model is a building block on which to construct a more realistic model.

3.2 Well-formedness Properties of Environment

The external world constitutes the environment in which the log manager must operate. By definition, the environment is constrained to respect certain conventions which govern its relationship with the log manager. These conventions are expressed in terms of the well-formedness properties presented in this section.

Let β denote a behavior for the log manager module, and let π_i represent the i^{th} action of β (where $i \in \mathcal{N}$ and $i \geq 1$). The behavior β is well-formed if and only if it satisfies the four properties expressed below.

WF1: $\forall x$: π_i=COMMIT$_x$ $\Longrightarrow$ $\nexists j$, $j{\neq}i$, such that π_j=COMMIT$_x$.

WF2: $\forall x$: π_i=ERASE$_x$ $\Longrightarrow$ $\exists j$, $j{<}i$, such that π_j=ERASABLE$_x$.

WF3: $\forall x$: π_i=ERASE$_x$ $\Longrightarrow$ $\nexists j$, $j{\neq}i$, such that π_j=ERASE$_x$.

WF4: $\forall x$: π_i=ERASABLE$_x$ $\Longrightarrow$ $\exists j$, $j{>}i$, such that π_j=ERASE$_x$.

Property WF1 states that the external world may commit a particular update, x, at most once. Property WF2 states that the external world cannot perform an ERASE$_x$ action until after the log manager has given it permission to do so via the ERASABLE$_x$ action, while WF3 constrains the external world to perform at most one ERASE$_x$ action for any particular DLR x. Finally, WF4 insists that the external world must eventually perform an ERASE$_x$ action in response to an ERASABLE$_x$ action.

These well-formedness properties which the environment must preserve can be represented in an automaton, called `ENV`, as shown in Figure 3.2.

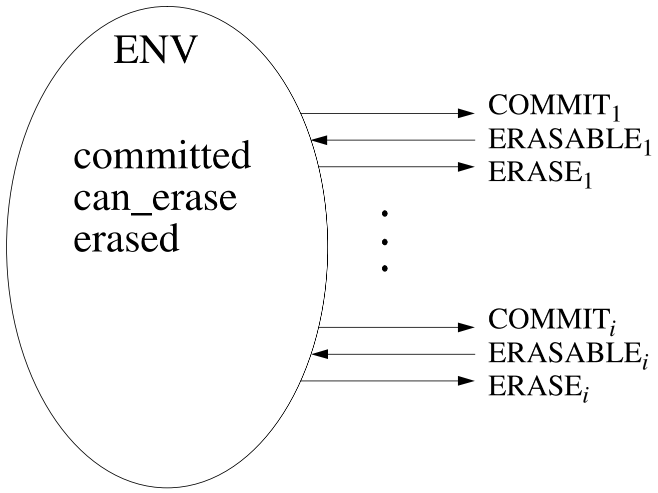


Figure 3.2: Automaton to Model Well-formed Environment

The names, types and initial values of the three variables which constitute the state of `ENV` are[1]:

variable	type	initial value
committed	$2^{\mathcal{N}}$	$\emptyset$
can_erase	$2^{\mathcal{N}}$	$\emptyset$
erased	$2^{\mathcal{N}}$	$\emptyset$

The `ENV` module has the following transition relation:

[1]$\mathcal{N}$ denotes the set of natural numbers {0,1,2,...}. For any set $\mathcal{S}$, $2^{\mathcal{S}}$ denotes the powerset of $\mathcal{S}$.

$\texttt{ERASABLE}_i$

Effect: can_erase $\leftarrow$ can_erase $\cup \{i\}$

$\texttt{COMMIT}_i$

Precondition: $i \notin$committed

Effect: committed $\leftarrow$ committed $\cup \{i\}$

$\texttt{ERASE}_i$

Precondition: ($i\in$can_erase) $\wedge$ ($i\notin$erased)

Effect: can_erase $\leftarrow$ can_erase $- \{i\}$

erased $\leftarrow$ erased $\cup \{i\}$

3.3 Specification of Correctness (Safety)

This section defines a very simple I/O automaton, called `SLM`, which embodies the safety property required of any implementation of XEL. Namely, it ensures that the record from the most recently committed update is not erased unless the records from all previously committed updates have already been erased. To prove this property, this section states a set of invariants which describe the composition of the `ENV` and `SLM` automata in all reachable states and then uses these invariants to prove that all behaviors of the `SLM` automaton satisfy the safety property.

3.3.1 I/O Automaton Model

Figure 3.3 illustrates the I/O automaton for the very simple version of the log manager. This automaton shall be referred to as `SLM`.

Three variables comprise the state of `SLM`. Their names, types and initial values are[2]:

variable	type	initial value
keep	$\mathcal{N}_\perp$	$\perp$
let_erase	$2^{\mathcal{N}}$	$\emptyset$
wait_erase	$2^{\mathcal{N}}$	$\emptyset$

The `SLM` automaton has the following transition relation:

[2] For any set $\mathcal{S}$, $\mathcal{S}_\perp$ denotes the *lifted domain* $\mathcal{S}_\perp = \mathcal{S} \cup \perp$, where $\perp$ is some unique bottom element that does not belong to $\mathcal{S}$.

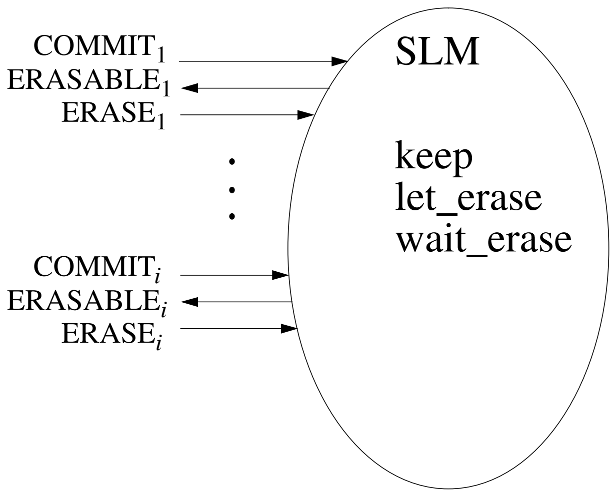


Figure 3.3: Specification Automaton for LM

$\texttt{COMMIT}_i$

Effect: if ((let_erase=$\emptyset$) AND (wait_erase=$\emptyset$))
  let_erase $\leftarrow$ let_erase $\cup$ $\{i\}$
 else
  if (keep$\neq\perp$)
   let_erase $\leftarrow$ let_erase $\cup$ {keep}
  keep $\leftarrow$ i

$\texttt{ERASE}_i$

Effect: wait_erase $\leftarrow$ wait_erase $-$ $\{i\}$
 if ((keep$\neq\perp$) AND (let_erase=$\emptyset$) AND (wait_erase=$\emptyset$))
  let_erase $\leftarrow$ let_erase $\cup$ {keep}
  keep $\leftarrow$ $\perp$

$\texttt{ERASABLE}_i$

Precondition: $i \in$let_erase
Effect: let_erase $\leftarrow$ let_erase $-$ $\{i\}$
 wait_erase $\leftarrow$ wait_erase $\cup$ $\{i\}$

3.3.2 Invariants for Composition of SLM and ENV

The following invariants apply to the system composed from the SLM and ENV automata. It is easy to verify inductively that they are true in all reachable states of the system.

Invariant 3.1

(keep=$\perp$) $\vee$ (let_erase$\neq\emptyset$) $\vee$ (wait_erase$\neq\emptyset$)

Invariant 3.2
$\forall x,\ x{\in}\mathcal{N},\ (\text{keep}{\neq}x) \vee (x{\in}\text{committed})$

Invariant 3.3
$\forall x,\ x{\in}\mathcal{N},\ (x{\notin}\text{let\_erase}) \vee ((\text{keep}{\neq}x) \wedge (x{\in}\text{committed}))$

Invariant 3.4
$\forall x,\ x{\in}\mathcal{N},\ (x{\notin}\text{wait\_erase}) \vee ((\text{keep}{\neq}x) \wedge (x{\notin}\text{let\_erase}) \wedge (x{\in}\text{committed}))$

Invariant 3.5
$(\perp{\notin}\text{let\_erase}) \wedge (\perp{\notin}\text{wait\_erase})$

3.3.3 Correctness of `SLM` Module

Theorem 3.4, at the end of this subsection, expresses XEL's important safety property: the log record from the most recently committed update is never erased before the records from all earlier committed updates have already been erased. Several supporting lemmas must first be proven. This subsection proves that the `SLM` automaton, when composed with `ENV`, satisfies this property. Throughout this section, α will represent an execution of the module composed of `SLM` and `ENV`, and π_i will represent the i^{th} action of α.

The following lemma states that after a particular update w has been committed, the `SLM` automaton keeps track of w in one of its three state variables at least until w is erased.

Lemma 3.1

$$\begin{array}{l} \quad \wedge \quad (\pi_l{=}\texttt{COMMIT}_w) \\ \quad \wedge \quad (\nexists m,\ l{<}m{\leq}j,\ \text{s.t. } \pi_m{=}\texttt{ERASE}_w) \\ \Longrightarrow \\ \quad \forall h,\ l{\leq}h{\leq}j,\ ((\text{keep}{=}w) \vee (w{\in}\text{let\_erase}) \vee (w{\in}\text{wait\_erase}))\text{ in state } t_h \end{array}$$

Proof:

- $\pi_l{=}\texttt{COMMIT}_w \Longrightarrow ((\text{keep}{=}w) \vee (w{\in}\text{let\_erase}))$ in state t_l
- $(((\text{keep}{=}w) \vee (w{\in}\text{let\_erase}) \vee (w{\in}\text{wait\_erase}))\text{ in state } t_{m-1}) \wedge (\pi_m{\neq}\texttt{ERASE}_w)$
  $\Longrightarrow ((\text{keep}{=}w) \vee (w{\in}\text{let\_erase}) \vee (w{\in}\text{wait\_erase}))$ in state t_m
- $\quad (((\text{keep}{=}w) \vee (w{\in}\text{let\_erase}))\text{ in state } t_l)$
  $\wedge \quad (\nexists m,\ l{<}m{\leq}j,\ \text{s.t. } \pi_m{=}\texttt{ERASE}_w)$
  $\Longrightarrow \forall h,\ l{\leq}h{\leq}j,\ ((\text{keep}{=}w) \vee (w{\in}\text{let\_erase}) \vee (w{\in}\text{wait\_erase}))$ in state t_h
  by induction

and thus the lemma has been proven. $\square$

The following lemma states that after a particular update x has been committed, then at least one of the let_erase and wait_erase state variables of the `SLM` automaton must be non-empty at least until x is erased (assuming that the execution is well-formed).

Lemma 3.2

$$\begin{array}{ll} & (\alpha \text{ is a well-formed execution}) \\ \wedge & (\pi_i{=}\mathtt{COMMIT}_x) \\ \wedge & (\exists k,\ k{>}i,\ \text{s.t. } \nexists j,\ i{<}j{\leq}k,\ \text{s.t. } \pi_j{=}\mathtt{ERASE}_x) \\ \Longrightarrow & \\ & \forall l,\ i{\leq}l{\leq}k,\ ((\text{let\_erase}{\neq}\emptyset) \vee (\text{wait\_erase}{\neq}\emptyset)) \text{ in state } \mathrm{t}_l \end{array}$$

Proof:

- $(\pi_i{=}\mathtt{COMMIT}_x) \wedge (\nexists j,\ i{<}j{\leq}k,\ \text{s.t. } \pi_j{=}\mathtt{ERASE}_x)$
  $\Longrightarrow \forall l,\ i{\leq}l{\leq}k,\ ((\text{keep}{=}x) \vee (x{\in}\text{let\_erase}) \vee (x{\in}\text{wait\_erase}))$ in state t_l
  by Lemma 3.1
- $\forall l,\ i{\leq}l{\leq}k,\ ((\text{keep}{=}\perp) \vee (\text{let\_erase}{\neq}\emptyset) \vee (\text{wait\_erase}{\neq}\emptyset))$ in state t_l
  by Invariant 3.1
- $\quad(((\text{keep}{=}x) \vee (x{\in}\text{let\_erase}) \vee (x{\in}\text{wait\_erase}))$ in state $\mathrm{t}_l)$
  $\wedge\ (((\text{keep}{=}\perp) \vee (\text{let\_erase}{\neq}\emptyset) \vee (\text{wait\_erase}{\neq}\emptyset))$ in state $\mathrm{t}_l)$
  $\Longrightarrow ((\text{let\_erase}{\neq}\emptyset) \vee (\text{wait\_erase}{\neq}\emptyset))$ in state t_l
- It therefore follows that
  $\forall l,\ i{\leq}l{\leq}k,\ ((\text{let\_erase}{\neq}\emptyset) \vee (\text{wait\_erase}{\neq}\emptyset))$ in state t_l
  and thus the lemma has been proven. □

The following lemma proves that, in a well-formed execution, a particular update x cannot be erased before it has been committed.

Lemma 3.3

$$\begin{array}{ll} & (\alpha \text{ is a well-formed execution}) \\ \wedge & (\pi_i{=}\mathtt{COMMIT}_x) \\ \wedge & (\pi_j{=}\mathtt{ERASE}_x) \\ \Longrightarrow & \\ & i{<}j \end{array}$$

Proof:

- $\pi_i{=}\mathtt{COMMIT}_x \Longrightarrow \nexists f,\ f{\neq}i,\ \text{s.t. } \pi_f{=}\mathtt{COMMIT}_x$ by WF1
- $\pi_j{=}\mathtt{ERASE}_x \Longrightarrow \exists h,\ h{<}j,\ \text{s.t. } \pi_h{=}\mathtt{ERASABLE}_x$ by WF2
- $\pi_h{=}\mathtt{ERASABLE}_x \Longrightarrow x{\in}\text{let\_erase}$ in state t_{h-1}
- $x{\in}\text{let\_erase}$ in state t_{h-1}
  $\Longrightarrow$ Either
  (1) $\exists f,\ f{\leq}h{-}1$, s.t.
  $$\begin{array}{ll} & (\pi_f{=}\mathtt{COMMIT}_x) \\ \wedge & (((\text{let\_erase}{=}\emptyset) \wedge (\text{wait\_erase}{=}\emptyset)) \text{ in state } \mathrm{t}_{f-1}) \end{array}$$

- $(\pi_f = \texttt{COMMIT}_x) \wedge (\nexists f,\ f \neq i, \text{ s.t. } \pi_f = \texttt{COMMIT}_x) \implies f = i$
- $(f = i) \wedge (f \leq h-1 < j) \implies i < j$

or

(2) $\exists g,\ g \leq h-1$, s.t. $\quad (\pi_g = \texttt{ERASE}_z \text{ for some } z)$
$\qquad\qquad\qquad\qquad \wedge \quad (\text{keep}=x \text{ in state } t_{g-1})$

- $\text{keep}=x$ in state $t_{g-1} \implies \exists f,\ f \leq g-1$, s.t. $\pi_f = \texttt{COMMIT}_x$
- $(\pi_f = \texttt{COMMIT}_x) \wedge (\nexists f,\ f \neq i, \text{ s.t. } \pi_f = \texttt{COMMIT}_x) \implies f = i$
- $(f = i) \wedge (f \leq g-1 < j) \implies i < j$

Therefore, the desired result follows for both possible cases and thus the lemma has been proven. □

The following theorem proves that, in a well-formed execution, the most recently committed update, x, cannot be erased before all previously committed updates have been erased.

Theorem 3.4

$$\begin{aligned}
& (\alpha \text{ is a well-formed execution}) \\
\wedge\ & (\pi_i = \texttt{COMMIT}_x) \\
\wedge\ & (\pi_j = \texttt{ERASE}_x) \\
\wedge\ & (\nexists k,\ i < k < j, \text{ s.t. } \pi_k = \texttt{COMMIT}_y \text{ for any } y) \\
\wedge\ & (\pi_l = \texttt{COMMIT}_w,\ l < i) \\
\implies & \\
& \exists m,\ m < j, \text{ s.t. } \pi_m = \texttt{ERASE}_w
\end{aligned}$$

Proof:

By contradiction. Assume $\nexists m,\ m < j$, s.t. $\pi_m = \texttt{ERASE}_w$

- $\pi_i = \texttt{COMMIT}_x \implies \nexists g,\ g \neq i$, s.t. $\pi_g = \texttt{COMMIT}_x$ — by WF1
- $(\pi_l = \texttt{COMMIT}_w) \wedge (l < i) \wedge (\nexists g,\ g < i, \text{ s.t. } \pi_g = \texttt{COMMIT}_x) \implies w \neq x$
- $(\pi_l = \texttt{COMMIT}_w) \wedge (\nexists m,\ m \leq j, \text{ s.t. } \pi_m = \texttt{ERASE}_w)$
  $\implies \forall h,\ l \leq h \leq j,\ ((\text{let\_erase} \neq \emptyset) \vee (\text{wait\_erase} \neq \emptyset))$ in state t_h
  — by Lemma 3.2
- $(\pi_i = \texttt{COMMIT}_x) \wedge (\pi_j = \texttt{ERASE}_x) \implies i < j$ — by Lemma 3.3
- $(l < i) \wedge (i < j) \wedge (\forall h,\ l \leq h \leq j,\ ((\text{let\_erase} \neq \emptyset) \vee (\text{wait\_erase} \neq \emptyset)) \text{ in state } t_h)$
  $\implies ((\text{let\_erase} \neq \emptyset) \vee (\text{wait\_erase} \neq \emptyset))$ in state t_{i-1}
- $(\pi_i = \texttt{COMMIT}_x) \wedge (((\text{let\_erase} \neq \emptyset) \vee (\text{wait\_erase} \neq \emptyset)) \text{ in state } t_{i-1})$
  $\implies \text{keep}=x$ in state t_i
- $\pi_l = \texttt{COMMIT}_w \implies \nexists n,\ n > l$, s.t. $\pi_n = \texttt{COMMIT}_w$ — by WF1
- $(\text{keep}=x \text{ in state } t_i) \wedge (x \neq w) \wedge (\nexists n,\ n > l, \text{ s.t. } \pi_n = \texttt{COMMIT}_w) \wedge (l < i)$
  $\implies \forall p,\ i \leq p,\ \text{keep} \neq w$ in state t_p
- $\pi_j = \texttt{ERASE}_x \implies \exists q,\ q < j$, s.t. $\pi_q = \texttt{ERASABLE}_x$ — by WF2
- $\pi_q = \texttt{ERASABLE}_x \implies x \in \text{let\_erase}$ in state t_{q-1}
- $x \in \text{let\_erase}$ in state $t_{q-1} \implies \exists r,\ r \leq q-1$, s.t. either

(1) $(\pi_r = \texttt{COMMIT}_x) \wedge (((\text{let\_erase}=\emptyset) \wedge (\text{wait\_erase}=\emptyset))$ in state $\text{t}_{r-1})$

- $(\pi_r = \texttt{COMMIT}_x) \wedge (\nexists g,\ g \neq i,$ s.t. $\pi_g = \texttt{COMMIT}_x) \Longrightarrow r = i$
- $r = i \Longrightarrow ((\text{let\_erase} \neq \emptyset) \vee (\text{wait\_erase} \neq \emptyset))$ in state t_{r-1}

  But this is a contradiction, so this case cannot be true.

or

(2) $\quad (\pi_r = \texttt{ERASE}_z$ for some z)

$\wedge \quad (((\text{keep}=x) \wedge (\text{let\_erase}=\emptyset) \wedge (\text{wait\_erase}=\{z\}))$ in state $\text{t}_{r-1})$

- $(\pi_r = \texttt{ERASE}_z) \wedge (r<q<j) \wedge (\nexists m,\ m<j,$ s.t. $\pi_m = \texttt{ERASE}_w) \Longrightarrow z \neq w$
- keep$=x$, $x \in \mathcal{N}$, in state $\text{t}_{r-1} \Longrightarrow \exists f,\ f \leq r-1,$ s.t. $\pi_f = \texttt{COMMIT}_x$
- $(\pi_f = \texttt{COMMIT}_x) \wedge (\nexists g,\ g \neq i,$ s.t. $\pi_g = \texttt{COMMIT}_x) \Longrightarrow f = i$
- $(f=i) \wedge (f \leq r-1) \Longrightarrow i \leq r-1$
- $(\pi_l = \texttt{COMMIT}_w) \wedge (\nexists m,\ m \leq j,$ s.t. $\pi_m = \texttt{ERASE}_w)$

  $\Longrightarrow \forall e,\ l \leq e \leq j,$

  $((\text{keep}=w) \vee (w \in \text{let\_erase}) \vee (w \in \text{wait\_erase}))$ in state t_e

  by Lemma 3.1
- $\quad (((\text{keep}=x) \wedge (\text{let\_erase}=\emptyset) \wedge (\text{wait\_erase}=\{z\}))$ in state $\text{t}_{r-1})$

  $\wedge \quad (x \neq w)$

  $\wedge \quad (l < i \leq r-1 \leq j)$

  $\wedge \quad (\forall e,\ l \leq e \leq j,$

  $((\text{keep}=w) \vee (w \in \text{let\_erase}) \vee (w \in \text{wait\_erase}))$ in state $\text{t}_e)$

  $\Longrightarrow z = w$

  But this is a contradiction, and so this case cannot be true either.

Since both possible cases must be false the original assumption must be false and thus the theorem has been proven. □

3.4 Implementation of Log Manager

This section describes an implementation of the log manager. It composes a set of constituent I/O automata to yield a module with the same external action signature as the `SLM` module. This log manager module shall be referred to as `LM`. To prove that `LM` satisfies XEL's safety property, this section states a set of invariants that characterize all reachable states when `LM` is composed with `ENV`, postulates a possibilities mapping f from the composition of `LM` and `ENV` to the composition of `SLM` and `ENV` and then proves that f is indeed a possibilities mapping. Given that `SLM` (when composed with `ENV`) is correct and that f is a possibilities mapping from `LM` to `SLM`, it immediately follows that `LM` (when composed with `ENV`) implements `SLM` and therefore satisfies XEL's safety property. Finally, this section proves a liveness property of `LM`: every record is eventually erased.

3.4.1 I/O Automata Model

Figure 3.4 depicts the composition of a collection of automata, all for the same object. The `LOT` automaton represents the object's LOT entry and the accompanying procedures which manage it. Each $\mathtt{DLR}_i$ automaton represents a different possible update to the object. Together, these automata compose the `LM` module which models the log manager's activity for the object. The external world issues a $\mathtt{COMMIT}_i$ command to instruct `LM` to commit update i. Some time later, `LM` sends an $\mathtt{ERASABLE}_i$ message to the external world to inform it that it is now allowed to erase the DLR from update i. In response, the outside world will eventually send an $\mathtt{ERASE}_i$ message back to `LM` to inform it that update i's DLR has been erased.

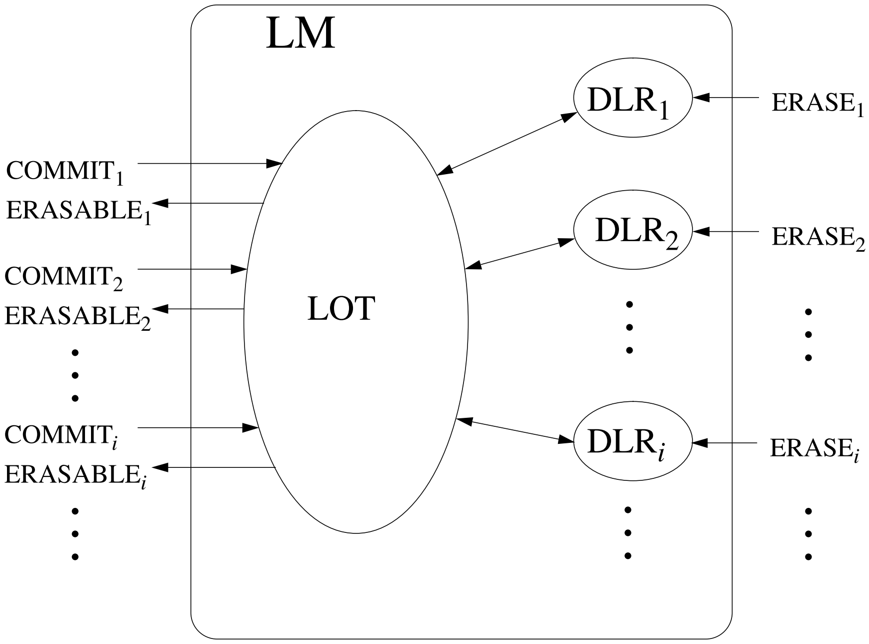


Figure 3.4: I/O Automata for an Object

This model ignores the problem of flow control between generations in a log stream. Management of the producer-consumer relationship between consecutive generations is an entirely different problem which is not considered here. The focus of this section is the management of each object's collection of REDO DLRs according to the state transition diagram that was represented in Figure 2.5. This state transition diagram implicitly takes into account the fact that each stream may have more than one generation, and that there may be more than one stream. It shall be proven that XEL guarantees a consistent state when DLRs are characterized by the state transition diagram of Figure 2.5. A more elaborate model that incorporates XEL's flow control activities would show that XEL preserves the state transition diagram for each object's DLRs. By implication, this more elaborate version of XEL must also guarantee a consistent state.

Action Signatures of Automata

The LOT automaton has the following action signature:

in	out	int
COMMIT$_i$	<ASSIGN$_i$,ts$_i$>	*none*
ACK_ASSIGN$_i$	CS_REQD$_i$	
ACK_CS_RECV$_i$	CS_RECV$_i$	
<DLR_GONE,ts$_i$>	ERASABLE$_i$	

In this action signature, as well as that given below for DLR$_i$, $i \in \mathcal{N}$ and ts$_i \in \mathcal{N}$, where $\mathcal{N}$ denotes the set of natural numbers.

Each DLR$_i$, $i \in \mathcal{N}$, automaton has the following action signature:

in	out	int
<ASSIGN$_i$,ts$_i$>	ACK_ASSIGN$_i$	*none*
CS_REQD$_i$	ACK_CS_RECV$_i$	
CS_RECV$_i$	<DLR_GONE,ts$_i$>	
ERASE$_i$		

After receiving a COMMIT$_i$ message from the external world, the LOT automaton chooses a unique timestamp, ts$_i$, for the associated DLR and sends an <ASSIGN$_i$,ts$_i$> message to DLR$_i$. The DLR$_i$ automaton receives the message and replies with an ACK_ASSIGN$_i$ message to the LOT.

The LOT automaton sends a CS_REQD$_i$ message to DLR$_i$ to instruct it to change its status from *unflushed* to *required*. Similarly, the CS_RECV$_i$ message informs DLR$_i$ that it should change its status to *recoverable*. DLR$_i$ does not bother to acknowledge receipt of a CS_REQD$_i$ message, but it does send an ACK_CS_RECV$_i$ message to acknowledge a previous CS_RECV$_i$ message.

After DLR i has been erased, its DLR$_i$ automaton sends a <DLR_GONE,ts$_i$> message to the LOT to inform it that the DLR whose timestamp was ts$_i$ no longer exists.

The subscripted messages from the LOT to DLR$_i$ (namely, <ASSIGN$_i$,ts$_i$>, CS_REQD$_i$ and CS_RECV$_i$) denote point-to-point communication. The fact that the <DLR_GONE,ts$_i$> message is not subscripted reflects the implementation of XEL, in which only the timestamp of an erased DLR is communicated to the LOT.

States of Automata

The following variables constitute the state of the `LOT` automaton[3]:

variable	type
pending_ts_assign	$2^{\mathcal{N} \times \mathcal{N}}$
recv_tss	$2^{\mathcal{N}}$
current_ts	$\mathcal{N}$
curr_reqd_ts	$\mathcal{N}_\perp$
curr_reqd_dlr	$\mathcal{N}_\perp$
curr_reqd_acked	$\mathcal{B}$
send_cs_reqd	$\mathcal{N}_\perp$
send_cs_recv	$2^{\mathcal{N}}$
pending_erasable	$2^{\mathcal{N}}$

Every member of pending_ts_assign represents a committed DLR for which no <$\mathtt{ASSIGN}_i$,ts_i> message has yet been generated. The recv_tss variable represents the set of timestamps which correspond to all DLRs whose status is merely recoverable. The `LOT` maintains a counter, current_ts, which it will assign as the timestamp value for the next committed DLR. The curr_reqd_ts and curr_reqd_dlr variables indicate the timestamp and identity, respectively, of the DLR which currently has status *required*, if there is such a DLR. In conjunction with these two variables, the curr_reqd_acked variable indicates if that DLR's automaton has acknowledged receipt of its timestamp. The send_cs_reqd variable represents the DLR to which a $\mathtt{CS\_REQD}_i$ message should be sent, if any such DLR exists. Likewise, send_cs_recv identifies all DLRs to which $\mathtt{CS\_RECV}_i$ messages should be sent. Finally, the pending_erasable set indicates all DLRs for which the `LOT` can issue an $\mathtt{ERASABLE}_i$ message.

The following variables constitute the state of each $\mathtt{DLR}_i$ automaton:

variable	type
$status_i$	{UNFL,REQD,RECV,NONR}
$pending\_ack_i$	$\mathcal{B}$
$timestamp_i$	$\mathcal{N}_\perp$

The $status_i$ variable of $\mathtt{DLR}_i$ indicates the current status of the DLR and may have one of the four values listed in the table above. If $pending\_ack_i$ is true, then $\mathtt{DLR}_i$ owes a message of response to the `LOT`; the value of $status_i$ determines the particular type of the message. The $timestamp_i$ variable represents a DLR's unique timestamp, and receives its value in response to an <$\mathtt{ASSIGN}_i$,ts_i> message from the `LOT`.

The variables which comprise the state of each of the constituent automata of the `LM` and the relationships amongst these automata are depicted in Figure 3.5.

[3] $\mathcal{B}$ denotes the set of boolean values {T,F}. For any sets $\mathcal{S}$ and $\mathcal{T}$, $\mathcal{S} \times \mathcal{T}$ denotes the set which is their cartesian product.

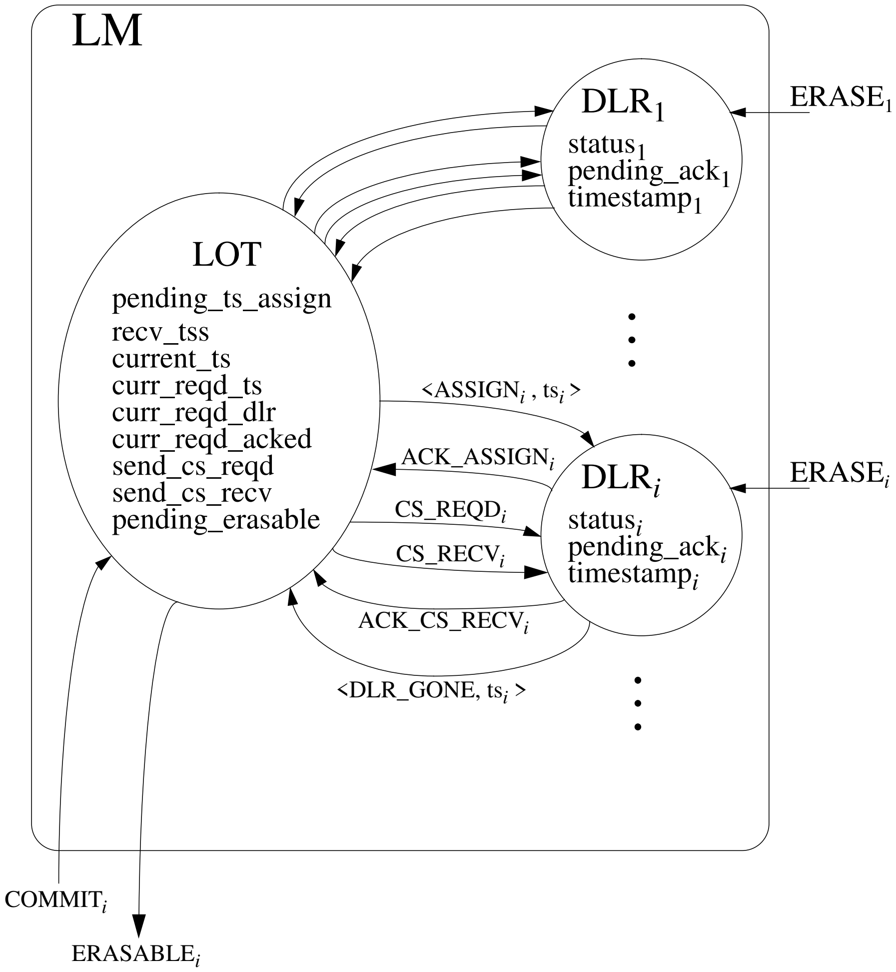


Figure 3.5: States and Action Signatures of Automata in LM Module

Initial State of System

The LOT automaton has a unique initial state. The initial values of the LOT's variables are:

variable	initial value
pending_ts_assign	$\emptyset$
recv_tss	$\emptyset$
current_ts	0
curr_reqd_ts	$\perp$
curr_reqd_dlr	$\perp$
curr_reqd_acked	F
send_cs_reqd	$\perp$
send_cs_recv	$\emptyset$
pending_erasable	$\emptyset$

Each $\texttt{DLR}_i$ automaton also has a unique initial state in which its variables have the following values:

variable	initial value
status_i	UNFL
pending_ack_i	F
timestamp_i	$\perp$

Transition Relations of Automata

The steps for the `LOT` automaton's input actions are as follows:

$\texttt{COMMIT}_i$

```
Effect: if (curr_reqd_ts≠⊥)
            recv_tss ← recv_tss ∪ {curr_reqd_ts}
            if (curr_reqd_acked=T)
                send_cs_recv ← send_cs_recv ∪ {curr_reqd_dlr}
        curr_reqd_ts ← current_ts
        curr_reqd_dlr ← i
        pending_ts_assign ← pending_ts_assign ∪ {<i,current_ts>}
        curr_reqd_acked ← F
        current_ts ← current_ts + 1
```

$\texttt{ACK\_ASSIGN}_i$

```
Effect: if (i=curr_reqd_dlr)
            curr_reqd_acked ← T
            if (recv_tss≠∅)
                send_cs_reqd ← i
              else
                recv_tss ← recv_tss ∪ {curr_reqd_ts}
                send_cs_recv ← send_cs_recv ∪ {i}
                curr_reqd_dlr ← ⊥
                curr_reqd_ts ← ⊥
          else
            send_cs_recv ← send_cs_recv ∪ {i}
```

$\texttt{ACK\_CS\_RECV}_i$

```
Effect: pending_erasable ← pending_erasable ∪ {i}
```

$<\texttt{DLR\_GONE},\text{ts}_i>$

```
Effect: if ((recv_tss={ts_i}) AND (curr_reqd_dlr≠⊥)
                    AND (curr_reqd_acked=T))
            recv_tss ← recv_tss ∪ {curr_reqd_ts}
            send_cs_recv ← send_cs_recv ∪ {curr_reqd_dlr}
            curr_reqd_dlr ← ⊥
            curr_reqd_ts ← ⊥
        recv_tss ← recv_tss − {ts_i}
```

The steps for the `LOT` automaton's output actions are specified below:

$\texttt{ERASABLE}_i$
- Precondition: $i \in$pending_erasable
- Effect: pending_erasable $\leftarrow$ pending_erasable $- \{i\}$

$<\texttt{ASSIGN}_i$, ts$_i>$
- Precondition: $<i$,ts$_i>\in$pending_ts_assign
- Effect: pending_ts_assign $\leftarrow$ pending_ts_assign $- \{<i$,ts$_i>\}$

$\texttt{CS\_REQD}_i$
- Precondition: i=send_cs_reqd
- Effect: send_cs_reqd $\leftarrow \perp$

$\texttt{CS\_RECV}_i$
- Precondition: $i \in$send_cs_recv
- Effect: send_cs_recv $\leftarrow$ send_cs_recv $- \{i\}$

The steps for the $\texttt{DLR}_i$ automaton's input actions are:

$<\texttt{ASSIGN}_i$, ts$_i>$
- Effect: timestamp$_i \leftarrow$ ts$_i$
  pending_ack$_i \leftarrow$ T

$\texttt{CS\_REQD}_i$
- Effect: if (status$_i$=UNFL)
  status$_i \leftarrow$ REQD

$\texttt{CS\_RECV}_i$
- Effect: status$_i \leftarrow$ RECV
  pending_ack$_i \leftarrow$ T

$\texttt{ERASE}_i$
- Effect: status$_i \leftarrow$ NONR
  pending_ack$_i \leftarrow$ T

The steps for the $\texttt{DLR}_i$ automaton's output actions are:

$\texttt{ACK\_ASSIGN}_i$
- Precondition: status$_i$=UNFL
  pending_ack$_i$=T
- Effect: pending_ack$_i \leftarrow$ F

$\texttt{ACK\_CS\_RECV}_i$
- Precondition: status$_i$=RECV
  pending_ack$_i$=T
- Effect: pending_ack$_i \leftarrow$ F

<`DLR_GONE`,ts_i>

Precondition: $status_i$=NONR
pending_ack$_i$=T
timestamp$_i$=ts$_i$

Effect: pending_ack$_i \leftarrow$ F

3.4.2 Invariants for Composition of LM and ENV

The following invariants will assist in the proof of `LM`'s correctness. They apply to the system composed of the `LM` and `ENV` modules. It is straightforward to verify that the invariants are true in the initial state of the system. Likewise, the definitions for the automata's actions ensure that the invariants remain true in all reachable states of the system.

For convenience, define a predicate recvbl(x), $x \in \mathcal{N}$, which characterizes a particular update, x, as recoverable or not, in a particular state of `LM`:

recvbl(x) $\equiv$ ((<x,u>$\in$pending_ts_assign for some u)
$\vee$ ((timestamp$_x \neq \perp$) $\wedge$ (status$_x \neq$NONR)))

Similarly, it is notationally convenient to define three other predicates. These predicates apply to a state of the `LM` automaton, but they have an obvious correspondence to the variables which comprise the state of the `SLM` automaton. The lm_ prefix is a reminder of the fact that they apply to the `LM` automaton. These predicates will play an important role in defining and proving a possibilities mapping from the states of `LM` to the states of `SLM`.

lm_keep(x) $\equiv$ (curr_reqd_dlr=x, $x \in \mathcal{N}$) $\wedge$ ($\exists y$, $y \neq x$, s.t. recvbl(y))

lm_let(x) $\equiv$ ((curr_reqd_dlr=x, $x \in \mathcal{N}$) $\wedge$ ($\nexists y$, $y \neq x$, s.t. recvbl(y)))
$\vee$ ((curr_reqd_dlr$\neq x$)
$\wedge$ (recvbl(x))
$\wedge$ ((status$_x \neq$RECV)
$\vee$ (pending_ack$_x \neq$F)
$\vee$ ($x \in$pending_erasable)))

lm_wait(x) $\equiv$ (status$_x$=RECV) $\wedge$ (pending_ack$_x$=F) $\wedge$ ($x \notin$pending_erasable))

The following invariants for the composition of `LM` and `ENV` are expressed in terms of the predicates defined above.

Invariant 3.6

$\forall x$, $x \in \mathcal{N}$, (status$_x$=UNFL) $\vee$ (status$_x$=REQD) $\vee$ (curr_reqd_dlr$\neq x$)

Invariant 3.7

$$\begin{array}{lll} \forall x,\ x\in\mathcal{N}, & & (\text{timestamp}_x\in\mathcal{N}) \\ & \vee & (\quad (\text{status}_x=\text{UNFL}) \wedge (x\neq\text{send\_cs\_reqd}) \\ & & \quad \wedge\ (x\notin\text{send\_cs\_recv}) \wedge (\text{pending\_ack}_x=\text{F}) \\ & & \quad \wedge\ (x\notin\text{can\_erase}) \wedge (x\notin\text{pending\_erasable})) \end{array}$$

Invariant 3.8

$$\begin{array}{lll} \forall x,\ x\in\mathcal{N}, & & (\text{curr\_reqd\_dlr}\neq x) \\ & \vee & (\exists v,\ v\in\mathcal{N},\ \text{s.t.} \quad (\text{curr\_reqd\_ts}=v) \\ & & \qquad \wedge\ (\quad (\text{timestamp}_x=v) \\ & & \qquad\quad \vee\ (<x,v>\in\text{pending\_ts\_assign}))) \end{array}$$

Invariant 3.9

$$\begin{array}{ll} \forall x,\ x\in\mathcal{N}, & \\ & (\nexists v\ \text{s.t.}\ <x,v>\in\text{pending\_ts\_assign}) \\ \vee & (\quad (\exists v,\ v\in\mathcal{N},\ \text{s.t.} \quad (<x,v>\in\text{pending\_ts\_assign}) \\ & \qquad\qquad \wedge\ (\nexists u,\ u\neq v,\ \text{s.t.}\ <x,u>\in\text{pending\_ts\_assign})) \\ & \quad \wedge\ (\text{timestamp}_x=\perp) \\ & \quad \wedge\ (\text{status}_x=\text{UNFL})) \end{array}$$

Invariant 3.10

$$\begin{array}{ll} \forall x,\ x\in\mathcal{N}, & \\ & (x\in\text{committed}) \\ \vee & (\quad (\text{curr\_reqd\_dlr}\neq x) \wedge (\nexists u\ \text{s.t.}\ <x,u>\in\text{pending\_ts\_assign}) \\ & \quad \wedge\ (x\neq\text{send\_cs\_reqd}) \wedge (x\notin\text{send\_cs\_recv}) \\ & \quad \wedge\ (x\notin\text{pending\_erasable}) \wedge (x\notin\text{can\_erase}) \\ & \quad \wedge\ (\text{pending\_ack}_x=\text{F}) \wedge (\text{timestamp}_x=\perp)) \end{array}$$

Invariant 3.11

$\forall x,\ x\in\mathcal{N},\ (\text{curr\_reqd\_dlr}\neq x) \vee ((x\notin\text{send\_cs\_recv}) \wedge (x\notin\text{can\_erase}))$

Invariant 3.12

$\forall x,\ x\in\mathcal{N},\ (x\notin\text{pending\_erasable}) \vee ((\text{pending\_ack}_x=\text{F}) \wedge (x\notin\text{can\_erase}))$

Invariant 3.13

$\forall x,\ x\in\mathcal{N},\ (x\notin\text{can\_erase}) \vee (\text{pending\_ack}_x=\text{F})$

Invariant 3.14

$\forall x,\ x\in\mathcal{N},\ (\text{status}_x=\text{RECV}) \vee ((x\notin\text{pending\_erasable}) \wedge (x\notin\text{can\_erase}))$

Invariant 3.15

$\forall x,\ x \in \mathcal{N},\ (x \notin \text{send\_cs\_recv}) \vee (\text{status}_x = \text{UNFL}) \vee (\text{status}_x = \text{REQD})$

Invariant 3.16

$$
\begin{aligned}
\forall x,\ x \in \mathcal{N},\quad & ((\nexists v \text{ s.t. } <x,v> \in \text{pending\_ts\_assign}) \wedge (\text{timestamp}_x = \perp)) \\
\vee\quad & (\exists v,\ v \in \mathcal{N}, \text{ s.t.} \\
& \quad ((<x,v> \in \text{pending\_ts\_assign}) \vee (\text{timestamp}_x = v)) \\
& \quad \wedge\ (\forall y,\ y \neq x,\quad (\quad (\nexists u \text{ s.t. } <y,u> \in \text{pending\_ts\_assign}) \\
& \qquad\qquad \wedge\ (\text{timestamp}_y = \perp)) \\
& \qquad \vee\ (\exists u,\ u \in \mathcal{N}, \text{ s.t.} \\
& \qquad\qquad (\quad (<y,u> \in \text{pending\_ts\_assign}) \\
& \qquad\qquad\quad \vee\ (\text{timestamp}_y = u)) \\
& \qquad\qquad \wedge\ (u \neq v))) \\
& \quad \wedge\ (v < \text{current\_ts}))
\end{aligned}
$$

Invariant 3.17

$$
\begin{aligned}
\forall x,\ x \in \mathcal{N}, & \\
& (\text{curr\_reqd\_dlr} = x) \\
\vee\ & (\neg \text{recvbl}(x)) \\
\vee\ & (\exists v,\ v \in \mathcal{N}, \text{ s.t.} \quad ((<x,v> \in \text{pending\_ts\_assign}) \vee (\text{timestamp}_x = v)) \\
& \qquad \wedge\ (v \in \text{recv\_tss}))
\end{aligned}
$$

3.4.3 Proof of Safety for LM

Let s be a state in the LM module (i.e., the implementation), t be a state in the SLM module (i.e., the specification), and let f denote a possibilities mapping from the states of LM to the states of SLM. This possibilities mapping f is defined as follows.

Definition 3.1

$$
\begin{aligned}
\forall x,\ x \in \mathcal{N}, & \\
& ((\text{lm\_keep}(x) \text{ in state s}) \quad \wedge \quad (\text{keep} = x \text{ in state t})) \\
\vee\ & ((\text{lm\_let}(x) \text{ in state s}) \quad \wedge \quad (x \in \text{let\_erase in state t})) \\
\vee\ & ((\text{lm\_wait}(x) \text{ in state s}) \quad \wedge \quad (x \in \text{wait\_erase in state t})) \\
\vee\ & (\quad (\neg \text{recvbl}(x) \text{ in state s}) \\
& \quad \wedge\ (((\text{keep} \neq x) \wedge (x \notin \text{let\_erase}) \wedge (x \notin \text{wait\_erase})) \text{ in state t})) \\
\Longleftrightarrow & \\
& t \in f(s)
\end{aligned}
$$

Refer to Section A.1 in Appendix A for all the lemmas and theorems which prove that f is a possibilities mapping from LM to SLM.

3.4.4 Proof of Liveness

Theorem A.69, which is found in Section A.2, states an important liveness property for LM: the DLR for every committed update is eventually erased. This property is formally expressed as:

$$\begin{array}{ll} & (\alpha \text{ is a well-formed and fair execution}) \\ \wedge & (\pi_i = \texttt{COMMIT}_x) \\ \Longrightarrow & \\ & \exists j,\ j > i, \text{ s.t. } \pi_j = \texttt{ERASE}_x \end{array}$$

where α represents an execution of the module composed of LM and ENV, and π_i represents the i^{th} action of α.

Refer to Section A.2 for the proof of this theorem and the many lemmas that support it.

Chapter 4

Parallel Logging

4.1 Parallel XEL

The XEL algorithm presented in Chapter 2 can be applied in a parallel system in which there are multiple *log streams*, each of which accepts incoming log records and operates independently of other log streams. Collectively, the log streams provide the bandwidth required for an application's log information. The name for this practice is *parallel XEL*.

Each log stream accepts incoming log records from client transactions and manages them according to the XEL algorithm. The streams operate independently of one another, except for dependencies introduced by the status values for log records. The LOT and LTT tables are distributed across numerous processors in the parallel system so that they can each provide the necessary throughput for operations on them.

Each log stream is segmented into the same number of generations. Generation i is the same size for each stream, but different generations within each stream may still be of different sizes. The positions of the head and tail for generation i in one stream are completely independent of the head and tail positions for another stream's generation i. That is, the LM performs forwarding and recirculation at each log stream independently of such activity at other streams.

The abstraction of multiple log streams that operate independently of one another is well suited to a system which requires an arbitrarily large number of disk drives to provide the necessary bandwidth for log information. The LM can dedicate only one or some small fixed number of disk drives to each particular stream.

Figure 4.1 illustrates the situation for a LM that manages four log streams, each composed of two generations. Within each stream, non-garbage log records are forwarded or recirculated and garbage records are thrown away (no garbage pails are shown in this figure in order to reduce visual clutter). Updated objects whose DLRs are anywhere in any of the log streams may have their new values flushed to the disk version of the database.

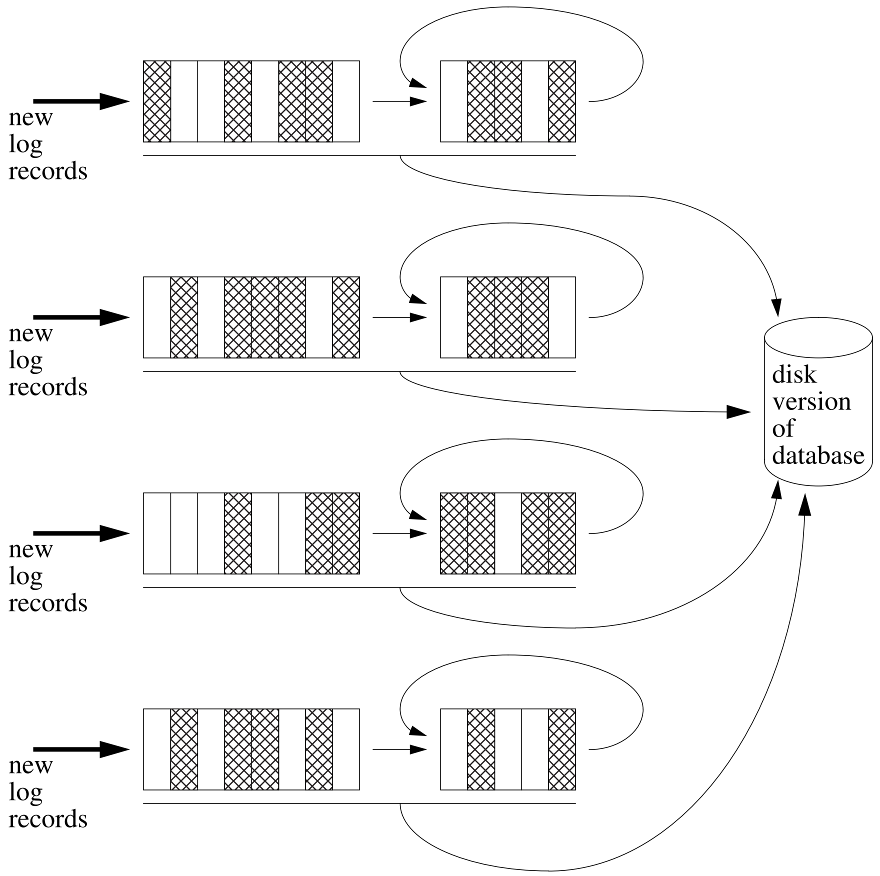


Figure 4.1: Four Parallel Log Streams

When the LM must examine or modify an object's LOT entry, it first hashes the object's oid to a processor identifier within the concurrent system and then it hashes the oid to a particular address within the processor's memory space. The number of processors over which the LOT is distributed must be sufficient to satisfy the throughput requirements of client transactions. Similarly, the LTT is implemented as a distributed hash table with a two-step translation procedure.

Each log stream has one particular processor that is responsible for managing its records[1]. The cells for a stream's relevant log records all reside in the memory space of the stream's processor. In general, the LM may send an object's DLRs to different log streams, and so the object's LOT entry cannot hold direct pointers to the cells for all

[1]This is not necessarily a one-to-one mapping. A processor may manage more than one log stream if it has sufficient processing power, memory capacity and communication bandwidth.

the object's relevant DLRs (as was the case for only a single log stream in Chapter 2). Rather, an object's LOT entry keeps track of the streams at which relevant DLRs exist. A *local LOT* at the processor of each of these streams, which is also associatively accessed via oid, holds pointers to the cells for each object's DLRs at the stream. Similarly, a transaction's LTT entry indicates the stream to which its `COMMIT` TLR, if any, was sent; the transaction's entry in a *local LTT* at that stream points to the cells for all copies of the `COMMIT` record within the stream.

As a special case, the LM may insist that all an object's DLRs go to the same stream, and the identity of this stream is determined by a function whose domain is the set of all oids. In this case, the LM can place the object's LOT entry at the processor that manages the stream to which its DLRs are sent so that indirection via a local LOT is unnecessary. This placement of LOT entries at processors that are responsible for managing log streams assumes that each processor has ample resources to support both purposes. Similarly, a transaction's LTT entry can be placed at the processor that manages the stream to which its `COMMIT` record will be sent so that indirection via the local LTT is eliminated (this assumes that each transaction is statically mapped to some stream which will receive its `COMMIT` record).

When only a single log stream exists, the LM adds a `COMMIT` TLR to the log (i.e., adds it to the buffer in main memory that currently holds records at the tail of generation 0) as soon as a transaction requests to commit. In the more general case of more than one log stream, the LM waits until all a transaction's DLRs are on disk before it generates a `COMMIT` TLR for it; this delay is expected to be less than 100 ms. Therefore, a transaction's `COMMIT` record marks both its intention and its eligibility to successfully terminate.

The synchronization between a transaction's DLRs and its `COMMIT` record is accomplished as follows. For each buffer of each log stream, the LM keeps a list of the transaction identifiers for all transactions which wrote log records to the buffer. For each transaction, the LM keeps a list which identifies the buffers to which the transaction has written log records; this list is stored in the transaction's LTT entry. Immediately after a buffer of log records has been written to disk, the LM examines the buffer's list of transactions. For each transaction in the list, the LM removes the buffer identifier from the list kept in the transaction's LTT entry. If this list becomes empty and the transaction is waiting to commit, then the LM generates a `COMMIT` record for the transaction; otherwise, the LM does nothing further and leaves the transaction waiting for the rest of the buffers on which it depends to be written to disk.

Crash recovery is almost the same as for the special case of only a single log stream, except that the POT and RTT data structures are distributed across processing nodes in a parallel machine so that they provide the necessary throughput. Like the LOT and LTT, they are implemented as distributed hash tables with a two step translation procedure. The RM's work at each log stream proceeds completely independently of recovery activity at other log streams. The RM sends each DLR to the processor that manages the POT entry for the object indicated in the DLR. Likewise, after retrieving a transaction's `COMMIT` record from disk, the RM sends it to the processor that is responsible for its RTT entry.

Many of the messages used in parallel XEL are quite short, typically only 10 to 20 Bytes in length. Low overhead interprocessor communication is therefore particularly important. For best performance, parallel XEL should be implemented on a fine-grain concurrent computer that provides low overhead, low latency communication primitives. The MIT J-Machine [11, 12, 48] is an existing example of such a machine. XEL will perform satisfactorily on other concurrent systems in which the overhead for interprocessor communication and synchronization is higher as long as the added delays are still relatively short compared to the delays for writing blocks to disk, the interconnection network provides sufficient bandwidth and CPU cycles are plentiful.

4.2 Three Different Distribution Policies

When a client transaction submits a log record to the LM, the LM must choose the stream(s) to which it will assign the record. The LM's *distribution policy* governs its choice of log streams for records. In general, copies of a log record may be sent to any number of streams. All the policies examined in this thesis send a log record to only one log stream.

This section proposes three distribution policies: *partitioned*, *random* and *cyclic*. These policies are all *oblivious* policies: they do not use information about current load[2] imbalances to help choose the stream to which to send a log record. More elaborate *adaptive* policies, which monitor load imbalances between streams and attempt to send records to streams so as to counteract current imbalances, are beyond the scope of this thesis.

The analyses in the following subsections consider only the bandwidth required for incoming log records to generation 0 of each log stream. The bandwidth for forwarded and recirculated records within each stream is ignored because it defies accurate analytical modelling. In practice, the bandwidth required for forwarded and recirculated records ought to be relatively small compared to that required for incoming new records.

4.2.1 Partitioned Distribution

The *partitioned* policy assigns each object to a particular log stream. A function whose only argument is an oid defines this mapping. For each object, the LM directs all its

[2] In these discussions, a log stream's load is defined to be the bandwidth demanded of it.

DLRs to the stream prescribed by this mapping. Similarly, another mapping (via some other function) from tid to stream number determines the stream to which the LM sends each transaction's `COMMIT` record.

The partitioned distribution is susceptible to "static skew" effects. Even if all objects are updated with the same frequency, some streams may be assigned more objects than others, and so they must provide more bandwidth for log information. By definition, the entire LM fails when it must refuse to accept a log record from a client transaction. The failure of only one log stream condemns the entire system, according to this definition. A log stream will fail when the bandwidth demanded of it exceeds its available bandwidth. Therefore, the maximum demanded bandwidth, over all log streams, is an important metric when evaluating a parallel logging system.

A quantitative analysis for a simple case can provide some insight into the static skew effect that threatens the *partitioned* distribution policy. Suppose that there are N objects and 2 log streams; denote the log streams as A and B. Each of the N objects is randomly assigned to a particular log stream; the probability that it is assigned to stream A is $P_A=0.5$ (and hence, $P_B=0.5$ is the probability that it is assigned to stream B instead) and is independent of the assignments for the other objects. Let $\mathbf{M}$ be a random variable[3] that denotes the maximum number of objects assigned to either stream. The probability distribution function for $\mathbf{M}$ is

$$\mathrm{Prob}[\mathbf{M}=m] = \begin{cases} 0 & \text{if } m < \frac{N}{2} \\ \binom{N}{\frac{N}{2}} \frac{1}{2^N} & \text{if } N \text{ is even and } m = \frac{N}{2} \\ 2\binom{N}{m} \frac{1}{2^N} & \text{if } m > \frac{N}{2} \end{cases}$$

It follows that the expected value for $\mathbf{M}$ is given by

$$\begin{aligned} \mathrm{E}[\mathbf{M}] &= \sum_{m=\lceil\frac{N}{2}\rceil}^{N} m \;\mathrm{Prob}[\mathbf{M}=m] \\ &= \begin{cases} \frac{1}{2^N}\left\{ \frac{N}{2}\binom{N}{\frac{N}{2}} + 2\sum_{m=1+\frac{N}{2}}^{N} m \binom{N}{m} \right\} & \text{if } N \text{ is even} \\ \frac{1}{2^{N-1}} \sum_{m=\lceil\frac{N}{2}\rceil}^{N} m \binom{N}{m} & \text{if } N \text{ is odd} \end{cases} \end{aligned}$$

[3] All random variables are typeset in boldface to emphasize their nature.

Define E[**M**]/N to be the load imbalance; it represents the expected fraction of the objects assigned to the stream with the most objects. Ideally, E[**M**]/N remains constant at 0.5 (i.e., each stream gets half the objects) for all N. Figure 4.2 plots E[**M**]/N as N increases. For small values of N, the imbalance between the two streams is quite pronounced. For example, E[**M**]/N=0.6 for N=16. That is, 60% of the objects go to one stream and the other 40% go to the other stream, on average; the first stream's bandwidth is 50% higher than that of the second stream. However, a significant imbalance remains even for fairly large N. For N=128, E[**M**]/N=0.535193 and so the busier stream's bandwidth is 15% higher than that of the other stream, on average. The total number of objects in a database may be quite high (several million, say) but a relatively small number of "hot" objects may receive a disproportionately large number of updates. If these hot objects are not evenly distributed across all the log streams, the static skew effect may lead to significant load imbalances. The smaller the set of hot objects and the higher their "temperature", the worse the threat of load imbalances becomes.

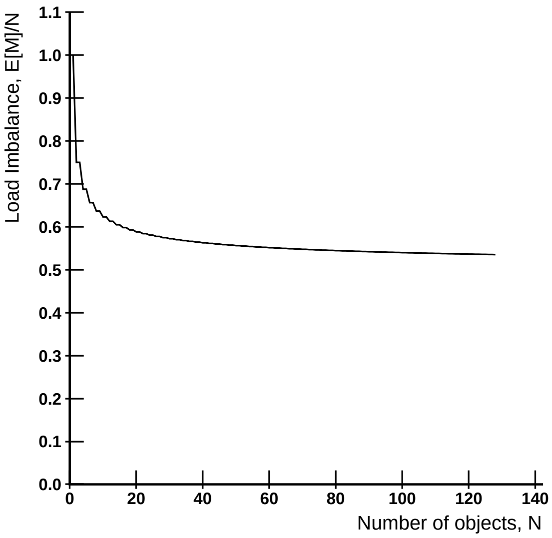


Figure 4.2: Load Imbalance vs. Number of Objects

Now consider what happens when the number of log streams increases in proportion to the number of objects. This is a reasonable exercise because each object may be characterized by a particular bandwidth (which depends on how often it is updated). If every object has the same characteristic bandwidth and this parameter remains constant, then a database's total demanded bandwidth is proportional to the number of objects in the database. To satisfy this demanded bandwidth, the LM must provide a number of log streams that is at least proportional to the number of objects.

Let there be N objects and S log streams. Each object is randomly assigned to one stream; the probability that the object is assigned to stream i is $\frac{1}{S}$, for $1 \leq i \leq S$. Let $P_{eq}(m,N,S)$ denote the probability that the maximum number of objects assigned to any stream is exactly m and $P_{ge}(m,N,S)$ denote the probability that the maximum number of objects assigned to any stream is at least m.

Lemma 4.1 $(N_2{>}N_1) \wedge (\lceil \frac{N_1}{S} \rceil{<}m{\le}N_2) \implies \mathrm{P}_{ge}(m,N_2,S){>}\mathrm{P}_{ge}(m,N_1,S)$

Proof:

Partition the collection of N_2 objects into two disjoint sets of sizes N_1 and $N_R{\equiv}N_2{-}N_1$. To assign the N_2 objects to the S streams, first assign the N_1 objects of the first set to streams and then assign the remaining N_R objects to streams. After assigning the first N_1 objects, let the random variable $\mathbf{B}_i$ denote the number of objects assigned to stream i, for $1{\le}i{\le}S$, and $\mathbf{\Psi}$ denote the set of streams that have a maximum number of objects assigned to them. That is, $\forall j{\in}\mathbf{\Psi}$, $\nexists k$, $1{\le}k{\le}S$, such that $\mathbf{B}_k{>}\mathbf{B}_j$. The probability that any one of the remaining N_R objects is assigned to one of the streams in $\mathbf{\Psi}$ is at least $\frac{1}{S}$ (this minimum occurs for the case of $N_R{=}1$ and $|\mathbf{\Psi}|{=}1$). Therefore,

$$\mathrm{P}_{ge}(m,N_2,S) \ge \mathrm{P}_{eq}(m{-}1,N_1,S)(\tfrac{1}{S}) + \mathrm{P}_{ge}(m,N_1,S)$$

and

$$\mathrm{P}_{eq}(m{-}1,N_1,S){>}0 \qquad \text{for } \lceil \tfrac{N_1}{S} \rceil{\le}m{-}1{\le}N_1$$

so

$$\tfrac{1}{S}\mathrm{P}_{eq}(m{-}1,N_1,S){>}0 \qquad \text{for } \lceil \tfrac{N_1}{S} \rceil{<}m{\le}N_1{+}1$$

and hence

$$\mathrm{P}_{ge}(m,N_2,S){>}\mathrm{P}_{ge}(m,N_1,S) \qquad \text{for } \lceil \tfrac{N_1}{S} \rceil{<}m{\le}N_1{+}1.$$

For larger values of m,

$$\mathrm{P}_{ge}(m,N_1,S){=}0 \text{ and } \mathrm{P}_{ge}(m,N_2,S){>}0 \qquad \text{for } N_1{+}1{<}m{\le}N_2$$

and so

$$\mathrm{P}_{ge}(m,N_2,S){>}\mathrm{P}_{ge}(m,N_1,S) \qquad \text{for } N_1{+}1{<}m{\le}N_2.$$

Therefore, the general result follows:

$$\mathrm{P}_{ge}(m,N_2,S){>}\mathrm{P}_{ge}(m,N_1,S) \qquad \text{for } \lceil \tfrac{N_1}{S} \rceil{<}m{\le}N_2 \qquad \square$$

Theorem 4.2 $\forall m,\ \lceil \frac{N}{S} \rceil{<}m{\le}2N,\ \mathrm{P}_{ge}(m,2N,2S){>}\mathrm{P}_{ge}(m,N,S)$

Proof:

Divide the set of $2S$ log streams into two equal (and disjoint) sets, each of size S. Assign the $2N$ objects to streams in two steps. In step 1, randomly assign each object to one of the two sets. In step 2, assign each object to a particular stream within its set. In step 1, the two sets are equally likely to be chosen when assigning objects to sets; similarly, the streams within a set all have the same probability of being chosen in step 2. Therefore, this two step procedure implies that the probability that a particular object is assigned to a particular stream is the same for all streams, as required by the statement of the problem. After step 1, there are two possible outcomes:

(a) Both sets have been assigned exactly N objects each. Denote this outcome by A.

or

(b) One set has been assigned more than N objects. Denote this outcome by B, and let the random variable $\mathbf{M}_B$ represent the number of objects assigned to the set with more objects. It must be true that $\mathbf{M}_B > N$.

The probability of outcome A is

$$P_A = \binom{2N}{N} (\tfrac{1}{2})^{2N}$$

and the probability of outcome B is

$$P_B = 1 - P_A = 1 - \binom{2N}{N} \tfrac{1}{2^{2N}}.$$

Therefore,

$$\mathrm{P}_{ge}(m, 2N, 2S) \geq \binom{2N}{N} \tfrac{1}{2^{2N}} (2\alpha - \alpha^2) + (1 - \binom{2N}{N} \tfrac{1}{2^{2N}})\ \mathrm{P}_{ge}(m, \mathbf{M}_B, S)$$

where $\alpha \equiv \mathrm{P}_{ge}(m, N, S)$ for $\lceil \frac{N}{S} \rceil < m \leq N$.

By Lemma 4.1, $\mathrm{P}_{ge}(m, \mathbf{M}_B, S) > \alpha$ for $\lceil \frac{N}{S} \rceil < m \leq \mathbf{M}_B$ because $\mathbf{M}_B > N$. Therefore,

$$\begin{aligned} \mathrm{P}_{ge}(m, 2N, 2S) &> \binom{2N}{N} \tfrac{1}{2^{2N}} (2\alpha - \alpha^2) + (1 - \binom{2N}{N} \tfrac{1}{2^{2N}})\alpha \\ &= \alpha[1 + \binom{2N}{N} \tfrac{1}{2^{2N}} (1 - \alpha)] \\ &> \alpha \qquad \text{because } 0 < \alpha < 1 \end{aligned}$$

This completes the proof that

$$\mathrm{P}_{ge}(m, 2N, 2S) > \mathrm{P}_{ge}(m, N, S) \qquad \text{for } \lceil \tfrac{N}{S} \rceil < m \leq N.$$

Turning now to larger values of m,

$$\mathrm{P}_{ge}(m, 2N, 2S) > 0 \quad \text{but} \quad \mathrm{P}_{ge}(m, N, S) = 0 \qquad \text{for } N < m \leq 2N$$

and so it follows that

$$\mathrm{P}_{ge}(m, 2N, 2S) > \mathrm{P}_{ge}(m, N, S) \qquad \text{for } N < m \leq 2N.$$

Combining these results yields the final conclusion:

$$\mathrm{P}_{ge}(m, 2N, 2S) > \mathrm{P}_{ge}(m, N, S) \qquad \text{for } \lceil \tfrac{N}{S} \rceil < m \leq 2N.$$

This completes the proof of the theorem. □

Theorem 4.2 implies that the threat of load imbalances becomes increasingly severe as the number of log streams increases in proportion to the number of objects. Hence, a LM must increase the number of log streams superlinearly if it is to maintain the same probability of failure (i.e., overload) as the number of objects in the database increases (assuming that the average rate at which each object is updated remains constant).

4.2.2 Random Distribution

The *random* distribution policy randomly chooses the stream to which each log record is sent; all log streams are equally likely to be chosen. Static skew is not a problem with the random distribution policy because it does not assign log records to streams on the basis of oids or tids; log records for different updates to the same object may go to different streams.

Nevertheless, different streams may receive different numbers of log records simply as a consequence of the LM's random decisions. Imbalances between log streams are now attributed to "dynamic skew" because they arise from decisions that the LM makes during operation rather than from a static assignment of objects and transactions to log streams.

Suppose that L log records are to be distributed to S log streams. The probability that the LM chooses to send a particular log record to stream i, for $1 \leq i \leq S$, is $\frac{1}{S}$ and is independent of its choices for other log records. Let the random variable $\mathbf{R}_i$ denote the number of log records that the LM sends to stream i, for $1 \leq i \leq S$; $\mathbf{R}_i$ has a binomial distribution [6] with a mean $\mathrm{E}[\mathbf{R}_i]=\frac{L}{S}$ and a variance $\mathrm{V}[\mathbf{R}_i]=\frac{L(S-1)}{S^2}$. Define another random variable $\mathbf{\Theta}_i \equiv \frac{\mathbf{R}_i}{L}$ to be the fraction of log records sent to log stream i. Since $\mathbf{\Theta}_i$ is a linearly scaled version of $\mathbf{R}_i$, its mean is $\mathrm{E}[\mathbf{\Theta}_i]=\frac{\mathrm{E}[\mathbf{R}_i]}{L}=\frac{1}{S}$ and its variance is $\mathrm{V}[\mathbf{\Theta}_i]=\frac{\mathrm{V}[\mathbf{R}_i]}{L^2}=\frac{S-1}{S^2 L}$. By the Chebyshev Inequality[4], $\mathrm{Prob}[|\mathbf{\Theta}_i - \frac{1}{S}| \geq \varepsilon] \leq \frac{S-1}{S^2 L \varepsilon^2}$ and so

$$\lim_{L \to \infty} \mathrm{Prob}[|\mathbf{\Theta}_i - \frac{1}{S}| \geq \varepsilon] = 0.$$

Under the assumption that all log records are the same size, this result proves that load imbalances between log streams are expected to diminish as the number of log records increases. Therefore, dynamic skew is not expected to be a serious problem in a system that operates on a continuous basis for long durations (so that many log records are written).

4.2.3 Cyclic Distribution

The *cyclic* policy assigns log records to streams in a round robin manner: the LM assigns a total order to the log streams and directs successive records to successive streams in the order. The cyclic distribution policy does not suffer from either static or dynamic skew effects. If all log records have the same size, it guarantees an optimal load balance amongst the log streams.

Again, assume that L log records are to be distributed across S streams. According to the cyclic distribution policy, the LM sends log record j to stream $1+(j \bmod S)$, for $j \geq 0$, and so stream i receives $l_i=\lceil \frac{L}{S} \rceil$ records if $i \leq (L \bmod S)$ and $l_i=\lfloor \frac{L}{S} \rfloor$ records otherwise, for $1 \leq i \leq S$. Define $\theta_i \equiv \frac{l_i}{L}$ to be the fraction of log records sent to stream i Note that $\theta_S \leq \theta_i \leq \theta_1$, for all i, $1 \leq i \leq S$. Under the assumption that all log records are the same size, θ_i is proportional to the load on stream i and so the load of the most

[4]The Chebyshev Inequality [6] states that for any random variable, $\mathbf{X}$, which is characterized by an expectation $\mathrm{E}[\mathbf{X}]=\mu$ and a variance $\mathrm{V}[\mathbf{X}]=\sigma^2$, the following relation is true: $\mathrm{Prob}[|\mathbf{X} - \mu| \geq \varepsilon] \leq \frac{\sigma^2}{\varepsilon^2}$ where ε is any arbitrarily chosen positive constant.

heavily loaded stream cannot exceed that of the most lightly loaded stream by more than a factor of $\frac{\theta_1}{\theta_S} = \frac{l_1}{l_S} \leq (1+\lfloor\frac{L}{S}\rfloor)/\lfloor\frac{L}{S}\rfloor$ but this quantity monotonically approaches 1.0 as L increases. Therefore, load imbalances amongst the streams are expected to be negligible after a system has been running for a while and many log records have been generated.

In practice, not all log records will be the same size so the cyclic policy no longer guarantees an optimal load balance amongst the set of log streams. Nevertheless, the cyclic policy is expected to yield reasonably good load balancing behavior for most applications if a system is allowed to run for a sufficiently long amount of time. The cyclic policy will serve as a touchstone against which to judge the other distribution policies.

The cyclic policy poses implementation problems as a system's degree of concurrency increases. The most straightforward implementation employs a single variable that keeps track of the stream to which the most recent log record was written. This single variable may become a serial bottleneck at some point as the number of processing nodes increases, or it may introduce significant complexity (such as a combining tree [19] implementation, for example). A simpler approach is to divide processing nodes into disjoint sets and perform cyclic distribution amongst the members of each set; that is, each set adheres to its own cyclic distribution discipline independently of the other sets. In this latter approach, each set has a separate variable that identifies the stream to which the most recent log record was written by any processing node in the set. The set size is restricted to some manageable limit, and the number of sets increases as a system's degree of concurrency increases. The superimposed loads from the different sets will still yield even load balancing amongst the log streams. Buffering may still be a problem, though. If the separate sets inadvertently "synchronize" so that they all send their records to the same stream at approximately the same time, one stream may receive a flood of records while other streams are relatively idle.

Chapter 5

Management of Precommitted Transactions

5.1 The Problem

The problem of managing precommitted transactions and the transactions which depend on them becomes much more complicated in a highly concurrent database that has a collection of parallel log streams. The following example illustrates the crux of the problem.

Suppose a transaction *tx1* acquires a write lock on some object *Obj5* in a database, updates the object and then requests to commit. Assume that the REDO DLR from *tx1*'s update to *Obj5* is already on disk when *tx1* requests to commit. In response to *tx1*'s request, the LM generates a `COMMIT` record for *tx1* and adds the record to some log stream's current buffer in main memory. Recall that the LM does not write the buffer to disk right away. Rather, it waits for the arrival of enough log records from other transactions to fill up the buffer (or for a time limit to expire) and then writes the buffer to disk.

Now suppose that some other transaction, *tx2*, reads the updated value of *Obj5* before *tx1*'s `COMMIT` record has been written to disk. Thus *tx2* becomes dependent on *tx1*. When *tx2* later wants to commit, the LM must create a `COMMIT` record for *tx2* and add it to some log stream's current buffer. If the `COMMIT` record for *tx2* goes to the same log stream as *tx1*'s `COMMIT` log record did, then there are no problems. Either these two log records belong to the same block of records, or *tx2*'s record is subsequently written to disk in another block. Either way, it is impossible for the RM to find *tx2*'s `COMMIT` record on disk, but not that of *tx1*.

Now suppose that *tx2*'s `COMMIT` log record is directed to a different stream than *tx1*'s record. It is possible that the buffer holding *tx2*'s `COMMIT` log record will fill up before the buffer to which *tx1*'s `COMMIT` record belongs. Let *buf1* and *buf2* denote the buffers that hold the `COMMIT` records from *tx1* and *tx2*, respectively. If *buf2* is written to disk but then a crash occurs before *buf1* can be written, the RM is faced with a problem. It finds a `COMMIT` record for *tx2* but not for *tx1*. How is it to know that *tx2* depended on *tx1*, so that its changes must be undone, despite the fact that its `COMMIT` record was written to disk?

The problem to be addressed concerns the management of precommitted transactions in a highly concurrent system which has many parallel log streams. The LM must ensure that the log on disk always contains sufficient information so that the RM can restore the database to a consistent state after a crash. If a crash occurs before a precommitted transaction commits, then the effects of this transaction and all transactions which depend on it must not be present in the restored database.

5.2 Shortcomings of Previous Approaches

Other researchers [15] have previously suggested that the LM ensure that buffers are written to disk in an order that will never jeopardize the consistency of the database. Transactions' `COMMIT` log records do not contain any explicit information about dependencies amongst transactions, so the LM must not allow the `COMMIT` log record of a transaction to be written to disk before any of the `COMMIT` log records for earlier transactions on which it depends.

Consider the application of this approach to the situation described in the previous section. The `COMMIT` record for transaction *tx1* is waiting in buffer *buf1* and the `COMMIT` record for *tx2*, which depends on *tx1*, is waiting in buffer *buf2* at a different log stream. The log manager must enforce a topological ordering amongst these buffers so that *buf2* is written to disk after *buf1*.

This approach becomes awkward as more complex situations arise. For example, suppose that another transaction *tx3* becomes dependent on *tx2* and requests to commit before either *buf1* or *buf2* has been written to disk. If the `COMMIT` record for *tx3* is added to *buf1*, then a dependency cycle now exists in the topological ordering amongst buffers. Buffer *buf2* must be written after *buf1* because of *tx2*'s dependency on *tx1*, but *buf1* must be written after *buf2* because of *tx3*'s dependency on *tx2*. Neither buffer can be written to disk before the other without risking a possibly inconsistent state after recovery. This situation is illustrated in Figure 5.1. The interdependencies amongst buffers at different log streams can be represented as a dependency graph. An arc from node x to node y in the dependency graph indicates that buffer x must be written after buffer y. In Figure 5.1, *buf2* must be written after *buf1* because of the dependency introduced by *tx2*, but *buf1* must be written after *buf2* because of the dependency introduced by *tx3*.

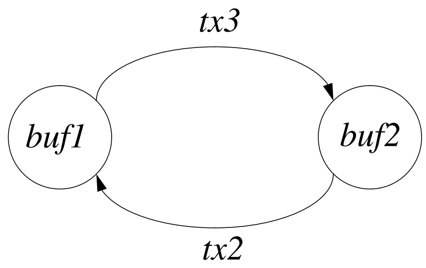


Figure 5.1: Deadlock in Dependency Graph for Buffers at Two Log Streams

One solution to this problem is to keep track of existing dependencies so that a cycle never forms. This leads to difficulties. In a large system with many parallel log streams, the maintenance of a dynamic dependency graph would entail prohibitive overhead. Before adding a `COMMIT` record to a stream's current buffer, the LM must traverse the graph to check that a cycle will not be created.

A static approach may involve less overhead. A static graph, defined at system initialization, specifies upon which other log streams a particular stream's buffers may depend. The graph is constructed so that no cycles can possibly occur. When a transaction's `COMMIT` record must be written, it is written to the stream which has the smallest set of allowed dependencies that includes all of the transaction's current dependencies. For any set of log streams, there must be a log stream which can have dependencies on all of them. This implies that the graph be a partial order with some unique bottom element. For example, the graph in Figure 5.2 is a suitable static partial ordering amongst seven log streams. For any set of nodes in the graph, there exists some node which is below all of them. Log stream *L7* is the unique bottom element.

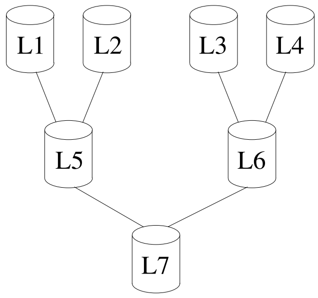


Figure 5.2: Static Dependency Graph of Log Streams

If a transaction has dependencies on buffers at log streams *L1* and *L2*, then its `COMMIT` record will be sent to log stream *L5*. A transaction with dependencies on *L2* and *L3* must have its `COMMIT` record directed to *L7*.

Although this static ordering reduces the overhead of maintaining a dependency graph to avoid cycles, it can lead to other problems. It restricts the LM's options for a distribution policy. Log streams at the bottom of the graph will tend to receive a higher load, at least in terms of `COMMIT` records, than streams near the top; this imbalance could persist indefinitely.

This section has explained the drawbacks of dynamic and static solutions to the problem of maintaining dependencies amongst buffers at different log streams so that transactions' `COMMIT` records are written to disk in an order that respects their dependencies. Dynamic approaches have significant run-time overhead, and static approaches are prone to load imbalances.

5.3 Logged Commit Dependencies (LCD)

This section presents a new solution, called *Logged Commit Dependencies* (LCD), that is an appealing alternative to the ones described in the previous section. All considerations about dependency graphs are banished. The choice of a log stream to which a record is written is no longer limited by synchronization constraints.

LCD introduces a new type of TLR called a `PRECOMMIT` record. When a transaction requests to commit, the LM immediately generates a `PRECOMMIT` record which explicitly identifies all the transaction's unsatisfied dependencies at the time of the request. The LM can send the `PRECOMMIT` record to any log stream and can write it to disk at any time.

Recovery becomes more complicated, however. If the LM wrote `PRECOMMIT` records but not `COMMIT` records (which are no longer absolutely necessary), the RM might be forced to unravel a deep "tree" of transaction dependencies before it can conclude that a recent transaction actually committed. To make the RM's job easier, the LM also generates a `COMMIT` record for each transaction after all the transaction's dependencies have been satisfied. This `COMMIT` record is simply an indication to the RM that the transaction did indeed commit before the crash occurred, and so it need not bother to check all the dependencies listed in the transaction's `PRECOMMIT` record.

The LM maintains a monotonically increasing *Log Sequence Number (LSN)* for each log stream and associates a unique LSN value with every block of records that it writes to disk at the stream. The LM places a block's LSN at the beginning of the buffer which has been allocated for it. When the LM decides to write the buffer to disk, it increments the stream's LSN and puts the new LSN at the beginning of the new current buffer.

Each transaction's LTT entry has a field, called *DLR_deps*, which keeps track of the transaction's dependencies on unwritten REDO DLRs. This *DLR_deps* field holds a set of pairs, where each pair has a stream identifier and a LSN. Whenever a transaction writes a REDO DLR to the log, the LM notes the stream to which it was sent and the

LSN for the buffer to which it was added; denote the stream as s and the LSN as n. The LM uses this information to update the transaction's LTT entry. If the transaction's *DLR_deps* field currently has no pair for stream s, it adds the pair $<s,n>$ to the set. Otherwise, it updates the existing pair for s so that it holds the new LSN n instead of its previous value. Similarly, the LM also maintains corresponding information for each buffer at each log stream. The LM keeps track of the current LSN for a buffer and the transactions that have had log records added to the buffer. After a buffer has been written to disk, the LM uses this information to update the appropriate LTT entries. Suppose that transaction t had a log record in a buffer at stream s whose LSN is n. After this buffer has been written to disk, the LM retrieves the current pair $<s,m>$ for stream s from the *DLR_deps* field in t's LTT entry and compares m to n. If $m=n$, then the LM removes the pair from the *DLR_deps* field (because t has not written any records to subsequent buffers at stream s); otherwise (i.e., $m>n$), it just leaves the current $<s,m>$ pair in t's *DLR_deps* field.

Each transaction's LTT entry also has three more fields, called *depends_on*, *is_depended_on_by* and *dep_tx_ctr*. A transaction t's *depends_on* field holds the set of transaction identifiers for all precommitted transctions on which transaction t depends, and t's *is_depended_on_by* field holds the set of transaction identifiers for all subsequent transactions that depend on t. The *dep_tx_ctr* field is an integer-valued counter that keeps track of the number of transactions on which t depends while t is in a precommitted state.

The LM must remember the identity of the precommitted transaction, if any, that most recently updated each object. This information is kept in each object's LOT entry. Whenever a transaction reads or updates an object, it becomes dependent on the precommitted transaction, if any, that previously updated the object. The LM adds this dependency information to the respective *depends_on* and *is_depended_on_by* fields in the LTT entries of both transactions.

Each `PRECOMMIT` record contains the following three fields:

txid:	identifier for the transaction that has requested to commit
dlr_streams:	<stream_id,LSN> pairs for all streams with unwritten DLRs
precomm_txs:	transactions on which this transaction depends

Suppose a transaction t requests to commit. The LM determines which REDO DLRs (for updates by t) are still waiting to be written to disk and which precommitted transactions (on which t depends) are still waiting to commit by examining the *DLR_deps* and *depends_on* fields, respectively, in transaction t's LTT entry. Unless both these fields are empty, the LM puts the contents of both fields into their corresponding fields of the `PRECOMMIT` record for t and sends the `PRECOMMIT` record to some log stream. For each object b that t modified (as indicated by the contents of the *obj_ids* set in t's LTT entry), the LM updates b's LOT entry to record the fact that its current value depends on precommitted transaction t and then the LM releases t's write lock on b. Finally, the

LM counts the number of transactions listed in the *depends_on* field of t's LTT entry and assigns this value to t's *dep_tx_ctr* counter. If *DLR_deps* and *depends_on* are both empty when t requests to commit, the LM just goes ahead and generates a `COMMIT` record for t.

Similar to before, a transaction t does not actually commit until all its DLRs have been written to disk, all the precommitted transactions on which it depends have committed, and its `PRECOMMIT` record (or its `COMMIT` record, as explained below) has been written to disk. Without the LCD technique and with no explicit dependency information in the `COMMIT` log record, transaction t committed at the instant that its `COMMIT` record was written to disk (since all dependencies had to be satisified before its `COMMIT` record could be written to disk). Now, the explicit information in the `PRECOMMIT` record allows the `PRECOMMIT` record to be written to disk before all dependencies on DLRs and earlier transactions have been satisfied. There may be some delay between the time that t's `PRECOMMIT` record is written to disk and the time that it commits.

The LM detects that a precommitted transaction t's last dependency (on either an unwritten DLR or a precommitted transaction) has been satisfied when both $DLR\_deps = \emptyset$ and $dep\_tx\_ctr = 0$ become true for the transaction. When this happens, the LM immediately generates a `COMMIT` record for t and sends it to any log stream; the LM can go ahead and generate a `COMMIT` record for t even before t's `PRECOMMIT` record has been written to disk. The transaction commits as soon as either its `PRECOMMIT` record or `COMMIT` record has been written to disk (and $DLR\_deps = \emptyset$ and $dep\_tx\_ctr = 0$).

When transaction t actually does commit, the LM sends an acknowledgement to t in response to its commit request, updates the LOT entries of all objects which t had modified, and updates the LTT entries for all transactions listed in the *is_depended_on_by* set in t's LTT entry. For each object that t modified (as indicated by the contents of the *obj_ids* set in t's LTT entry), the LM first checks to see if the object still depends on t. If so, it changes the LOT entry to indicate that the object no longer depends on any precommitted transaction. Otherwise, the object must now depend on some subsequent precommitted transaction, and so the LM does not change the LOT entry. The LM processes all the members of the *is_depended_on_by* field in t's LTT entry immediately after t commits. For each transaction u in this set, the LM decrements the *dep_tx_ctr* counter in u's LTT entry.

The pseudocode for the LM's management of precommitted transactions, using the LCD technique, is given below.

```
read_object(txid, object_id) {
    lot_entry ← LOT entry for object object_id
    ltt_entry ← LTT entry for transaction txid
    ptx ← lot_entry−>precom_tx
    if (ptx≠NULL) {
        pretx_ltt_entry ← LTT entry for transaction ptx
```

```
        pretx_ltt_entry->is_depended_on_by ←
                                  pretx_ltt_entry->is_depended_on_by ∪ {txid}
        ltt_entry->depends_on ← ltt_entry->depends_on ∪ {ptx}
    }
}


update_object(txid, object_id) {
    lot_entry ← LOT entry for object object_id
    ltt_entry ← LTT entry for transaction txid
    ptx ← lot_entry->precom_tx
    if (ptx≠NULL) {
        pretx_ltt_entry ← LTT entry for transaction ptx
        pretx_ltt_entry->is_depended_on_by ←
                                  pretx_ltt_entry->is_depended_on_by ∪ {txid}
        ltt_entry->depends_on ← ltt_entry->depends_on ∪ {ptx}
    }
    <stream,lsn> ← write_log_record(txid, object_id)
    if (∃x s.t. <stream,x> ∈ ltt_entry->DLR_deps) {
        ltt_entry->DLR_deps ← ltt_entry->DLR_deps − {<stream,x>}
    }
    ltt_entry->DLR_deps ← ltt_entry->DLR_deps ∪ {<stream,lsn>}
}


request_to_commit(txid) {
    ltt_entry ← LTT entry for transaction txid
    ltt_entry->tx_status ← precommitted
    if ((ltt_entry->DLR_deps=∅) AND (ltt_entry->depends_on=∅)) {
            generate_commit_rec(txid)
        }
        else {
            generate_precomm_rec(txid, ltt_entry->DLR_deps,
                                            ltt_entry->depends_on)
        }
    for (b ∈ ltt_entry->obj_ids) {
        lot_entry ← LOT entry for object b
        lot_entry->precom_tx ← txid
        release write lock on object b
    }
    ltt_entry->dep_tx_ctr ← |ltt_entry->depends_on|
}
```

```
tx_committed(txid) {
   send acknowledgement of commit to txid
   ltt_entry ← LTT entry for transaction txid
   ltt_entry−>tx_status ← committed
   for (b ∈ ltt_entry−>obj_ids) {
      lot_entry ← LOT entry for object b
      if (lot_entry−>precom_tx = txid) {
         lot_entry−>precom_tx ← NULL
      }
   }
   for (u ∈ ltt_entry−>is_depended_on_by) {
      pretx_committed(u)
   }
}


pretx_committed(txid) {
   ltt_entry ← LTT entry for transaction txid
   ltt_entry−>dep_tx_ctr ← ltt_entry−>dep_tx_ctr − 1
   if (         (ltt_entry−>tx_status = precommitted)
        AND   (ltt_entry−>DLR_deps=∅)
        AND   (ltt_entry−>dep_tx_ctr=0) ) {
      generate_commit_rec(txid)
      if (PRECOMMIT record from txid is already on disk) {
         tx_committed(txid)
      }
   }
}


buffer_written_to_disk(txid, stream, lsn) {
   ltt_entry ← LTT entry for transaction txid
   no_commit_yet ← (ltt_entry−>DLR_deps≠∅)
   ltt_entry−>DLR_deps ← ltt_entry−>DLR_deps − {<stream,lsn>}
   if (         (ltt_entry−>tx_status=precommitted)
        AND   (no_commit_yet)
        AND   (ltt_entry−>DLR_deps=∅)
        AND   (ltt_entry−>dep_tx_ctr=0) ) {
      generate_commit_rec(txid)
      if (PRECOMMIT record from txid is already on disk) {
         tx_committed(txid)
      }
   }
}
```

```
precommit_or_commit_record_written_to_disk(txid) {
   ltt_entry ← LTT entry for transaction txid
   if (          (ltt_entry−>tx_status=precommitted)
        AND   (ltt_entry−>DLR_deps=∅)
        AND   (ltt_entry−>dep_tx_ctr=0) ) {
      tx_committed(txid)
   }
}
```

The RM now has greater responsibilities. It may discover transaction t's `PRECOMMIT` record on disk, but it must do some detective work to figure out if t really did commit before the crash occurred. For every log stream listed in t's `PRECOMMIT` record, the RM must check that all records in the stream up to and including the LSN indicated in the `PRECOMMIT` record were written to disk prior to the crash. This is easily accomplished, by inspecting the LSN numbers in all the blocks found in the log on disk. Likewise, for every indicated precommitted transaction, it must verify that the transaction did indeed commit before the crash. The RM keeps track of these dependencies by using two new fields, called *depends_on* and *is_depended_on_by*, that belong to each transaction's RTT entry. These fields are analogous to their counterparts in the LTT.

Let τ_{max} denote the maximum time required to fill a buffer and write it to disk. A transaction will commit (and generate a `COMMIT` record) within time τ_{max} after it submits its commit request. Therefore, a `COMMIT` record exists in the log on disk for every transaction which precommitted at least $2\tau_{max}$ prior to a crash. At worst, the RM must deduce the fates of only those transactions which precommitted in the last $2\tau_{max}$ seconds prior to the crash. It is expected that this number of transactions will be small, compared to the total size of the log.

A transaction t's `PRECOMMIT` record explicitly lists all the streams on which t depends for its REDO DLRs to be written to disk. To reduce the size of the `PRECOMMIT` record, and thus save disk space and bandwidth, the LM can postpone generating a `PRECOMMIT` record for t until all its REDO DLRs have been written to disk. Of course, transaction t can still release all its write locks (after the DBMS has updated the objects' LOT entries to record their dependency on t) as soon as t requests to commit. After all t's REDO DLRs have been written to disk, the LM generates a `PRECOMMIT` record for t that lists all the precommitted transactions on which t still depends at the time the `PRECOMMIT` record is generated. Now, the RM may need to deduce the fates of transactions that precommitted as early as $3\tau_{max}$ prior to a crash.

Now consider how to integrate LCD into the XEL technique. The LM continues to manage DLRs exactly the same as before, but has more complexity for the handling of `PRECOMMIT` and `COMMIT` records. Each `PRECOMMIT` record has a status of either *required* or *recoverable*. A `PRECOMMIT` record is initially *required*, and becomes *recoverable* after the LM has written a `COMMIT` record (for the same transaction) to disk. The LM must keep a *required* PRECOMMIT record in the log, but can throw away a *recoverable* PRECOMMIT record at its earliest convenience.

The LM must retain a transaction t's COMMIT record until all subsequent transactions which depend on t have had COMMIT records written to the log. Suppose some transaction u, which depends on t, commits. As soon as u's COMMIT record is on disk, the LM retrieves the contents of the *depends_on* field from u's LTT entry. For each member v of this field, the LM removes u from the *is_depended_on_by* field in v's LTT entry. Since u depended on t, the LM will remove u from the *is_depended_on_by* field in t's LTT entry. As soon as the LM detects *is_depended_on_by*=$\emptyset$ in t's LTT entry, it concludes that no subsequent transactions require t's COMMIT record any longer. The LM changes the status of a transaction's COMMIT record to *recoverable* as soon as *is_depended_on_by*=$\emptyset$ for the transaction, no *recoverable* UNDO DLRs remain from the transaction, and any remaining REDO DLRs have only *recoverable* status (these latter two conditions are determined by maintaining a counter in each transaction's LTT entry, as described in Section 2.8).

LCD can increase the maximum throughput for any particular object in the database, but this enhanced performance comes at a cost. LCD requires more storage for extra fields in each LOT, LTT and RTT entry; these extra fields maintain dependency information. It also increases the complexity and computational requirements of the LM and RM. Finally, the PRECOMMIT records consume disk space and bandwidth.

Chapter 6

Experimental Results

This chapter evaluates XEL and the three proposed distribution policies for parallel logging. Disk space, disk bandwidth, main memory requirements and recovery time are the evaluation criteria throughout the chapter.

Section 6.1 describes the event-driven simulator by which the experiments were performed. It explains each of the input parameters, documents the fixed parameters, presents the definitions of XEL's data structures as expressed in the C programming languange [36] and justifies the validity of the simulation model.

Section 6.2 quantitatively evaluates and compares the performances of XEL and the FW technique for a single log stream as various application characteristics vary. Four sets of experiments consider separately the effects of: (1) the probability of long transactions, (2) the duration of long transactions, (3) the size of DLRs from long transactions and (4) the "data skew" which characterizes access patterns to objects in a database. The results of this section demonstrate that XEL can significantly reduce the amount of disk space required for log information, compared to FW, although XEL requires significantly more main memory and may entail increased disk bandwidth for log information. Recovery is I/O bound, so recovery time is less for XEL than for FW.

Section 6.3 quantitatively evaluates and compares the load balancing properties of the three oblivious distribution properties as the number of parallel log streams increases. Three separate sets of experiments consider the cases of low, moderate and high data skew, respectively. In these experiments, the log streams are managed by only the XEL technique (FW is no longer of interest in this section). This section's results indicate that all three policies yield approximately equal load balancing behavior for low data skew. The *partitioned* policy performs slightly worse than *random* and *cyclic* for moderate data skew, and it is much worse for high data skew. However, the *partitioned* policy generally consumes the least amount of main memory because it does not require indirection via the LOT and LTT data structures.

6.1 Simulation Environment

To quantitatively evaluate XEL and the three distribution policies for parallel logging, the author implemented an event-driven simulator. The simulator is written in C and runs on SPARCstations.

6.1.1 Input Parameters

The user can specify the following input parameters:

timestamps:	flag to indicate if disk version of database keeps timestamps
arrival rate:	rate at which transactions are initiated
tx pdf:	statistical mix of transaction types
object pdf:	statistical access pattern to objects
flush rate:	rate for flushing updates to disk version of database
generations:	number and sizes of generations
recirculation:	flag to turn recirculation on or off
num streams:	number of parallel log streams
distn policy:	distribution policy for parallel logging
runtime:	duration of simulated time span
recovery:	flag to request recovery after normal logging activity ends

The **timestamps** parameter specifies whether or not timestamps are assumed to exist in the disk version of the database. If timestamps exist, then the simulator uses the EL [35] algorithm; otherwise, it uses the more complicated (but more general) XEL algorithm.

The simulator initiates transactions at regular intervals, according to the specified arrival rate (transactions per second).

The user specifies an arbitrary number of different transaction types and their probability distribution function (pdf). For each type of transaction, the user states the probability of occurrence, the duration of execution, the number of REDO DLRs written and the size of each DLR. Figure 6.1 graphically represents this transaction model for a transaction that generates $N=2$ REDO DLRs in a system with only one log stream.

Whenever a new transaction must be initiated, the simulator randomly (according to the pdf) selects its type. After choosing its type, the simulator schedules when its REDO DLRs will be written. The DLRs are written at equally spaced intervals, with the last being written only some short time ϵ (equal to $t_3 - t_2$) prior to completion. Suppose that the transaction's lifetime (specified as part of its type) is T. It will finish execution and request to commit (at time t_3) T seconds after it started. Its last DLR

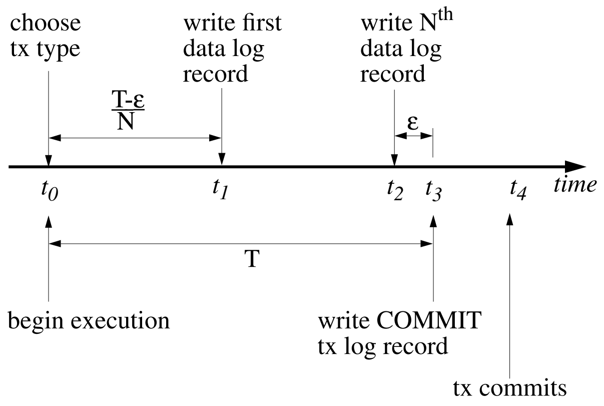


Figure 6.1: Simulation Transaction Model

is written (at time t_2) $T - \epsilon$ before it finishes, and each DLR is written $(T - \epsilon)/N$ after the preceding one, where N is the number of REDO DLRs written by a transaction of this type. After the LM has generated a `COMMIT` TLR for the transaction and sent it to a particular log stream, the transaction continues to wait for acknowledgement (at time t_4) from the LM before it actually commits; this delay occurs because the LM waits until a buffer is almost full before writing it to disk at the tail of generation 0, and then there is some delay τ_{Disk_Write} for transferring the contents to disk.

Whenever a transaction writes a REDO DLR, the simulator randomly picks some oid, according to the access probabilities specified by the user and subject to the constraint that the oid has not already been chosen for an update by a transaction which is still active. The set from which an oid can be chosen consists of all integers from 0 up to NUM_OBJECTS−1, where NUM_OBJECTS is the total number of objects (a fixed value). The user breaks up this set of objects into several classes. For each class, the user specifies the probability of occurrence and the size as a proportion of the total number of objects. When a DLR is to be written, the simulator first chooses a class according to the specified object pdf and then randomly selects an available oid from within this class.

To control the rate at which the CM can flush updates, the user specifies some number of disk drives and the time required to write a block to any of these drives. There can be at most one request at a time for any particular drive. The user can increase the maximum rate at which updates are flushed by increasing the number of drives or decreasing the time to write a block to any drive. The NUM_OBJECTS objects are striped evenly over these drives. That is, for D drives, object i is mapped to drive $i \bmod D$. Striping ensures that the objects within each class are distributed as evenly as possible across the different drives, so no drive is relatively overloaded by a large number of "hot spot" objects[1]. Each updated object requires a separate disk write (i.e., the simulator assumes that there is negligible locality of updates within a disk

[1] A "hot spot" object is one which is updated very frequently.

block). Each disk drive attempts to service pending flush requests in a manner that minimizes access time. The simulator assumes that the difference between two objects' oids corresponds to their locality on disk. For the purpose of calculating the difference between two oids, the simulator assumes that the sequence of integers assigned to their disk drive wraps around.

The user specifies the number of generations and the size (number of disk blocks) of each generation. The size of each disk block is fixed in the simulator.

In some experiments, it is worthwhile to examine the LM's behavior without recirculation in the last generation, just to see the effect of simply segmenting the log. There is an input flag to specify whether recirculation in the last generation is turned on or off. If recirculation is disabled and the LM cannot advance the tail of the last generation because it would overwrite a non-garbage log record at the head, then it refuses to accept any more incoming log records to the last generation; this tends to exert "back pressure" on younger generations.

The user can specify the number of log streams that are to operate in parallel. If more than one log stream is specified, then the user must also specify one of the three distribution policies.

If a log stream refuses to accept a log record, the simulator kills the client transaction that submitted the request. A more realistic simulator would stall, rather than kill the transaction. The current version of the simulator suffices because the experimental objective will be to determine the LM's resource requirements to support a particular load without needing to kill or stall transactions.

After simulating normal logging activity for the specified runtime, the simulator will also simulate recovery if the recovery flag has been set. Recovery uses the state of the log on disk in the condition which exists immediately after the specified runtime has elapsed. The simulator models only the first phase of recovery, in which the contents of the log streams are retrieved from disk and the most recently committed value in the log, if any, is determined for each object that had a DLR in the log. The second phase of recovery, in which the disk version of the database is updated with these values from the log, is not considered. In practice, this work can be performed in background after normal processing has resumed, so it is reasonable to ignore it when simulating recovery.

The current version of the simulator does not incorporate the LCD technique; all transactions retain their write locks until they commit. No experimental data are available for the performance of a LM which employs the LCD technique.

6.1.2 Fixed Parameters

Several parameters are fixed in the simulator. The delay ϵ between the write for a transaction's last REDO DLR and its request to commit is fixed at 1 ms. The capacity of each disk block is 2000 Bytes[2]. The LM attempts to keep $N_{free} \geq 3$ blocks available in each generation to hold incoming log records. Four disk block buffers (2048 Bytes each) are provided for generation 0 of each stream. Each `COMMIT` TLR is assumed to require 8 Bytes. The simulator conservatively assumes a fixed delay of τ_{Disk_Write}=15 ms to transfer a buffer's contents to disk when writing out records to the log. For each log stream, at most one disk write operation (to any generation) can be outstanding at any time. If several generations have buffers waiting to be written to them, the simulator gives older generations priority over generation 0 when it must schedule the next buffer to be written to disk. The simulator uses the *group commit* technique [5]; a log record is not written to disk until its buffer is as full as possible. Therefore, the delay between the time a record is added to a buffer and the time it is written to disk is generally longer than τ_{Disk_Write}. The number of objects in the database is fixed at NUM_OBJECTS=10^7. Disk I/O from each log stream is entirely sequential during recovery, so the simulator assumes that only 5 ms is required to retrieve a block from disk when reading the log. When recovering the contents of a block, each DLR requires 100 μs to process and a TLR requires 40 μs.

These fixed parameters are summarized in the following table:

Parameter	Value
ϵ = delay from last REDO DLR to commit request	1 ms
Capacity of each disk block in log	2000 Bytes
N_{free} = threshold number of free blocks per generation	3 blocks
Number of buffers (for generation 0) per log stream	4 buffers
Size of each `COMMIT` record	8 Bytes
τ_{Disk_Write} = delay to write a block to the log	15 ms
Maximum number of outstanding disk writes per stream	1 write
NUM_OBJECTS = number of objects in the database	10^7 objects
Delay to read a block from log during recovery	5 ms
Time required to process a DLR during recovery	100 μs
Time required to process a `COMMIT` TLR during recovery	40 μs

6.1.3 Data Structures

The following declarations define XEL's principal data structures for the most general situation of numerous parallel log streams and any distribution policy. They are expressed according to the syntax of the C programming language [36], in which the

[2]A block size of 2048 is typical, but the simulator assumes 48 Bytes are reserved for bookkeeping purposes and so only the remaining 2000 Bytes are available to hold log records.

simulator itself is written. These data structures are intended for a distributed memory message passing parallel system architecture, rather than a shared memory model.

```
struct str_lsm {               /* one log stream id in a list       */
  int    stream_id;            /* identifier of a log stream        */
  int    min_recov_tstamp;     /* needed for parallel XEL           */
  struct str_lsm  *next;       /* pointer to next cell in the list  */
};

struct str_oid {               /* one object id in a list of oids   */
  OBJID  oid;                  /* object identifier                 */
  struct str_oid  *next;       /* pointer to next cell in the list  */
};

struct str_ltt_entry {         /* LTT entry for a transaction       */
  TXNID     tid;               /* id of transaction                 */
  TX_STATUS status;            /* current status of the transaction */
  int       num_rqd_dlrs;      /* number of required DLRs remaining */
  int       rec_str;           /* log stream to which COMMIT written */
  struct    str_pcg       *set_cgs; /* cmt grps on which tx depends */
  struct    str_oid       *obj_ids; /* objects modified by this tx  */
  struct    str_ltt_entry  *next;    /* other txs in same hash buckt */
};

struct str_lot_entry {         /* LOT entry for an object           */
  OBJID  oid;                  /* id of object with records in log  */
  int    uncm_tstamp;          /* timestamp for most recent DLR     */
  int    comm_tstamp;          /* tstamp most recently committed DLR */
  struct str_lsm       *lstms; /* log streams with DLRs for object  */
  struct str_lot_entry *next;  /* other objects in same hash bucket  */
};

struct str_rel_cell {  /* cell to point to a relevant log record    */
  TXNID  tid;          /* id of associated transaction              */
  int    block_num;    /* index of block in log to which rec belongs */
  int    rec_length;   /* size of the log record (in Bytes)         */
  R_STAT rec_status;   /* current status of log record              */
  int    tstamp;       /* timestamp of update, if cell is for DLR   */
  struct str_llot_entry *p_obj; /* parent object, if cell is for DLR */
  struct str_rel_cell   *next;  /* more cells for same obj or tx    */
  struct str_rel_cell   *left;  /* left neighbor in doubly lkd list */
  struct str_rel_cell   *right; /* right neighbor in doubly lkd list */
};

struct str_llot_entry {         /* local LOT entry for an object    */
```

```
  OBJID  oid;                   /* id of object with DLRs in stream  */
  struct str_rel_cell   *cells; /* list of cells for object's DLRs   */
  struct str_llot_entry *next;  /* other objects in same hash bucket */
};

struct str_lltt_entry {         /* local LTT entry for a transaction */
  TXNID  tid;                   /* id of tx with TLRs in log         */
  struct str_rel_cell   *cells; /* list of cells for the tx's TLRs   */
  struct str_lltt_entry *next;  /* next tx in the hash bucket list   */
};

static struct str_lot_entry *lot_tbl[LOT_TBL_SIZE];  /* LOT hash tbl */
static struct str_ltt_entry *ltt_tbl[LTT_TBL_SIZE];  /* LTT hash tbl */
```

The `str_rel_cell` definition does not include a field that indicates a log record's type (TLR, REDO DLR or UNDO DLR). Such an extra field is unnecessary. The contents of the `rec_status` field identify both the type and the status of a record. For example, a *required* REDO DLR and a *required* UNDO DLR have different values in the `rec_status` fields of their cells.

The `str_lsm` and `str_lot_entry` structures go together on processing nodes that administer portions of the LOT. The `str_oid` and `str_ltt_entry` structures belong together on nodes that manage the LTT. The `str_llot_entry`, `str_lltt_entry` and `str_rel_cell` structures are used at nodes that manage the log streams.

Assume that each oid and tid requires 8 Bytes. Integers and pointers consume 4 Bytes each. Each field of type TX_STATUS or R_STAT requires 4 Bytes (these types are `typedef`'ed to `int`). The amount of storage required for each of these structures is summarized below.

Structure name	Storage required (Bytes)
`str_lsm`	12
`str_oid`	12
`str_ltt_entry`	32
`str_lot_entry`	24
`str_rel_cell`	40
`str_llot_entry`	16
`str_lltt_entry`	16

For the specific case of a single log stream or multiple log streams with the partitioned distribution strategy, indirection via the local LTT and local LOT is no longer necessary. The `cells` field from the `str_lltt_entry` structure replaces the `rec_str` field in the `str_ltt_entry` structure. Similarly, the `cells` field from the `str_llot_entry` structure replaces the `lstms` field in the `str_lot_entry` structure. The `p_obj` pointer in `str_rel_cell` now points to an instance of the `str_lot_entry` type. The declarations of the `str_lsm`, `str_llot_entry` and `str_lltt_entry` structure types can be omitted. The storage requirements for each of the `str_oid`, `str_ltt_entry`, `str_lot_entry` and `str_rel_cell` structure types remains unchanged, despite these modifications[3].

The simulator can also report the storage requirements for FW logging. The user must specify only a single log stream with a single generation, of course. The FW method requires a simpler version of the LTT. As before, each entry in this LTT corresponds to a particular transaction. It has fields for the transaction's identifier (8 Bytes), the current status of the transaction (4 Bytes), the number of DLRs still waiting to be flushed (4 Bytes), the position within the log of the first record written by the transaction (4 Bytes) and a pointer to the next entry in the LTT (4 Bytes). The FW method keeps track of a transaction until after it has committed and none of its updates need to be flushed to the disk version of the database. When a committed transaction no longer has any updates waiting to be flushed, the FW method removes it from the LTT. Therefore, the user should set the simulator's **timestamps** flag to *true* so that the EL algorithm is used[4], even though the disk version of the database may not actually keep a timestamp with every object in the database.

6.1.4 Validity of Simulation Model

The simulator provides sufficient flexibility to realistically evaluate various LM configurations for many different applications. It does not permit a user to precisely model every possible application, but it does allow a user to succinctly specify the characteristics for a broad range of applications. The simulator's inherent technological assumptions only approximate reality, yet they capture the important characteristics of the underlying technology while abstracting out many details that are largely irrelevant. Therefore, the simulator provides sufficient power to evaluate XEL and its parallel variants as important parameters vary.

The probabilistic transaction model statistically describes an application's static mixture of transactions. It is worthwhile to examine XEL's behavior as the relative lifetimes of different transaction types vary, and so the simulator provides this capability. The number and size of each transaction type's log records affect XEL's performance, and so a user can also vary these parameters. The probabilistic transaction model does not provide sufficient power to specify every possible application. For example, an application in which exactly every eighth transaction is 10.0 s long and the remaining applications have duration 1.0 s cannot be modelled, nor can an application whose transactions do not write REDO DLRs at equally spaced intervals. Despite these shortcomings, enough different application transaction mixes can be specified to provide

[3] For the case of only a single log stream, the `set_cgs` field can be removed from the `str_ltt entry` declaration, thus saving another 4 Bytes per transaction.

[4] The EL algorithm removes a committed transaction's LTT entry as soon as all its DLRs have become garbage.

meaningful results.

Likewise, the deterministic arrival rate enables a user to control the system's overall throughput, but limits the ability to control precisely when each transaction is initiated. A Markov arrival pattern [6], for example, cannot be accurately modelled with the current version of the simulator. However, variations in the arrival pattern are not an important issue for the evaluation of XEL; the overall throughput is the important parameter.

The simulator is an open system. It does not incorporate feedback in its scheduling of the times at which transactions are initiated, DLRs are generated and commits are requested. If a transaction manager refrains from initiating new transactions before it has received commit acknowledgements for enough previous transactions, then the open system assumption is unrealistic. However, the open system assumption can be justified for some other applications. As long as the LM continues to accept incoming log records, there may be little reason why the DBMS would change the rate at which it initiates new transactions. Likewise, a client transaction never needs to wait for a reply after requesting the LM to write a REDO DLR to the log; at the time that the request is made, the LM already knows whether or not it has space available on disk for the record and it can respond to the client transaction immediately. A transaction's lifetime is therefore largely unaffected by the performance of the LM, as long as the LM is able to accept its log records. After a transaction requests to commit, it must wait for acknowledgement from the LM. For the most general case of more than one log stream, the LM must make sure that all a transaction's DLRs are on disk before it generates a `COMMIT` record for the transaction, and then there is some additional delay before the `COMMIT` record arrives on disk. As the LM becomes busier, queues form and a buffer of records may need to wait longer before being written to disk. Therefore, the length of time that a transaction must wait for acknowledgement to its request to commit depends on the LM's load, but this delay does not affect the times at which the transaction writes its DLRs and requests to commit. The experimental objective will be to evaluate the LM's resource requirements such that the LM can accept all requests without needing to kill (or stall) any client transactions, so the open system assumption does not diminish the significance of the experimental results.

The probabilistic specification of data access patterns enables a user to model different collections of data objects in a database. These collections are characterized by the frequency with which transactions update their member objects. A user can model a wide range of different "data skews". Again, the statistical nature of this modelling is concise and simple but there are some applications whose exact data access patterns cannot be expressed in terms of this model. This inability to accurately model all possible applications does not prevent one from using the simulator to conduct experiments which illustrate important aspects of XEL's performance.

XEL's behavior is largely influenced by what happens as records approach the head of a generation, because only then does XEL decide whether or not to forward or recirculate a record. The arrival of subsequent records push a record toward the head of its generation so that XEL must decide its fate, but the exact times of arrival of these records are largely irrelevant. This observation supports the claim that the simulator's acknowledged inabilities to accurately model all possible applications' characteristics does not seriously limit its worth for the purposes of studying XEL.

The statistical specifications of transaction types and data access patterns are static, as is the arrival rate. In reality, an application's characteristics may vary over time. The simulator does not permit specification of dynamically varying parameters. Despite this shortcoming, meaningful experimental results may be obtained for important cases in which an application's parameters remain static. These results can provide valuable insights into XEL's behavior.

The simulator uses a simple model for disk I/O. A flush to the disk version of the database always requires the same duration (a parameter which the user specifies). In reality, this duration may vary from one write operation to the next. However, small fluctuations in the actual duration to flush an update will have only a minor effect on XEL's performance.

The simulator provides only a first order estimate of the bandwidth required for log information. It assumes that each block write operation to the log requires the same amount of time, τ_{Disk_Write}. In reality, the time required to write a block of log records to disk depends on whether the I/O is sequential or random in nature. Successive writes to the tail of generation 0 are sequential and so they will generally have a short duration (such as 5 to 10 ms each). When the LM must occasionally write a block to the tail of generation 1, for example, the resulting disk I/O is random; it will tend to take significantly longer (such as 20 ms) because of seek and rotational delays. The effects of the random disk I/O to all generations except generation 0 can be minimized by choosing generation 0 to be sufficiently large so that only a small fraction of log records need to be forwarded. Section 7.3.3 explains an optimization for a system with several log streams so that most disk I/O to log disks is sequential.

The simple model of the CM assumes that no uncommitted updates are ever written out to the disk version of the database. Hence, UNDO DLRs are never needed. Economic trends justify this simplification. For many applications, the savings in disk I/O bandwidth outweigh the price of the extra main memory needed to buffer uncommitted updates. For example, suppose that each transaction modifies X Bytes of state, is T_l seconds long and begins T_d seconds after the preceding transaction. The extra memory required to buffer all uncommitted updates from a continuous sequence of transactions is no more than XT_l/T_d Bytes[5]. If an UNDO DLR were written for each uncommitted update, the extra disk bandwidth would be X/T_d Bytes/sec. The technique introduced in [25] permits a comparison of the relative costs for these two options. Let C_m represent the cost (in dollars per Byte) of main memory and C_d represent the cost of disk

[5]To be more precise, the upper limit is actually $X\lceil T_l/T_d \rceil$, but the approximation XT_l/T_d suffices for a first order analysis.

bandwidth (in dollars per Byte/sec). The cost of buffering uncommitted updates is $C_m X T_l / T_d$, and the cost of writing UNDO DLRs is $C_d X / T_d$. Therefore, it is less expensive to buffer uncommitted updates in main memory if $T_l < C_d / C_m$. A typical disk drive costs at least \$200 and provides a maximum I/O bandwidth of 2 MBytes/sec, so $C_d = 10^{-4}$ \$·sec/Byte. On the other hand, 1 MByte of DRAM costs approximately \$20, so $C_m = 20 \times 10^{-6}$ \$/Byte. Hence, buffering of uncommitted updates is better economically if $T_l < 5$ sec. For many applications, transactions have lifetimes shorter than 5 sec. As the price of DRAM continues to fall, relative to the price of disk bandwidth, T_l will continue to increase.

The simulator concerns itself with the management of log information on disk and the associated data structures which must reside in main memory. It does not account for computational requirements nor for interprocessor communication. To some extent, the consumption and management of these latter two resources depend on the specific system upon which the logging and recovery system is implemented and so it is difficult to accurately account for them. Furthermore, computation and communication are relatively cheap and abundant in concurrent systems, and so they do not deserve nearly as much serious attention as disk I/O, which is a limited and relatively expensive resource.

6.2 Extended Ephemeral Logging for a Single Stream

This section presents the results of many experiments which were conducted to observe the behavior of XEL as applied to a single log stream and to understand the effects of varying different parameters. For comparison purposes, the traditional FW technique was simulated by specifying a single generation with no recirculation. The FW simulation did not involve any checkpointing activity; the firewall was always the oldest log record from the oldest transaction in the system. This omission favors FW because it ignores the overhead (in terms of disk space and bandwidth) associated with checkpointing.

There are several evaluation criteria. Disk space, disk bandwidth (in terms of block writes per second) and main memory requirements (for the LOT and LTT) are the main criteria for normal logging activity. Elapsed time is the primary criterion for recovery.

The following parameters are specified for all experiments, unless otherwise stated.

There are two types of transactions. The first is of 1.0 s duration and writes 2 DLRs, each of size 100 Bytes. The second lasts 10.0 s, in which time it writes 4 DLRs of size 100 Bytes each. Their probabilities of occurrence are 0.95 and 0.05, respectively.

The arrival rate is 100 TPS.

There is no data skew. That is, all NUM_OBJECTS objects are equally likely to be chosen whenever an update is to be performed.

To provide sufficient bandwidth for flushing updates, each experiment specifies 10 disk drives with a transfer time of 25 ms. The conservative 25 ms time allows for some read operations to be interspersed with writes.

All tests of XEL use two generations. The minimum possible sizes for these generations are determined experimentally. Recirculation is disabled, so that it is possible to assess the effect of simply segmenting the log. There is only one log stream.

The simulation time is 500 s; these results reflect the minimum disk space required to support 500 s of logging activity such that no transaction is killed.

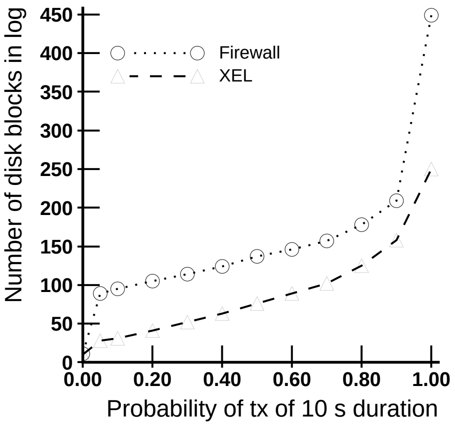
6.2.1 Effect of Transaction Mix

For the first set of experiments, the probabilities of occurrence for the two transaction types are varied. The probability of the long transaction type increases from 0 to 1.0, while the probability of the short type decreases accordingly.

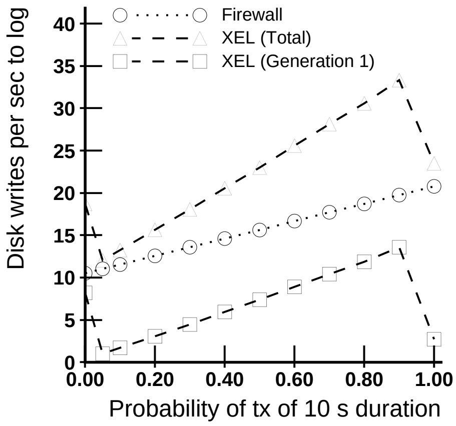
Figure 6.2(a) plots the disk space requirements (number of blocks) versus the transaction mix for both FW and XEL. The corresponding graphs of disk bandwidth (to only the log), main memory requirements and recovery time are shown in Figures 6.2(b) to 6.2(d), respectively.

XEL's advantages are most apparent for the 5% mix. It reduces disk space by a factor of 3.2 with only a 9.1% increase in bandwidth. XEL requires 13 times as much main memory as FW for the 5% mix, but this requirement is still modest in absolute terms; XEL needs only 57.5 KBytes of main memory. The time required to read in the log from disk dominates recovery time for both XEL and FW, so XEL offers much faster recovery. As the probability of 10 s transactions increases, XEL's relative advantage over FW diminishes. The reductions in disk space and recovery time are not as large, but the increase in bandwidth is greater.

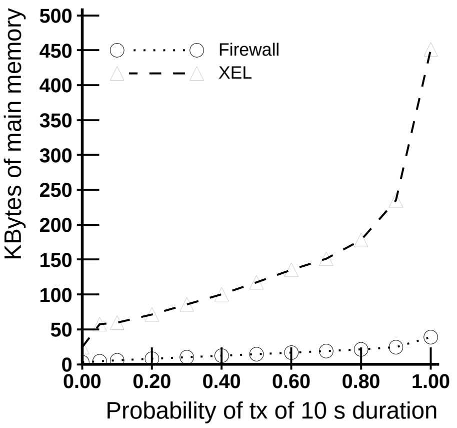
As the probability of the long transaction type approaches 1.0, the rate at which objects are updated approaches the maximum rate at which updates can be flushed (400 updates/s). The resultant queueing delay causes DLRs to tend to remain *unflushed* longer, and so the length of a single FW log increases accordingly. In the case of XEL, many of the DLRs have had their updates flushed by the time the LM must decide whether or not to forward them to generation 1; most of these DLRs have *recoverable* status and need not be forwarded. Only a fraction of all log records have *unflushed* or *required* status as they approach the head of generation 0, and so only these records advance into generation 1. By throwing away the garbage records at the head of generation



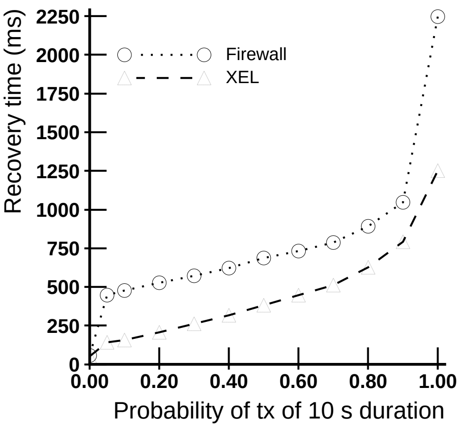
(a) Disk Space



(b) Disk Bandwidth



(c) Main Memory



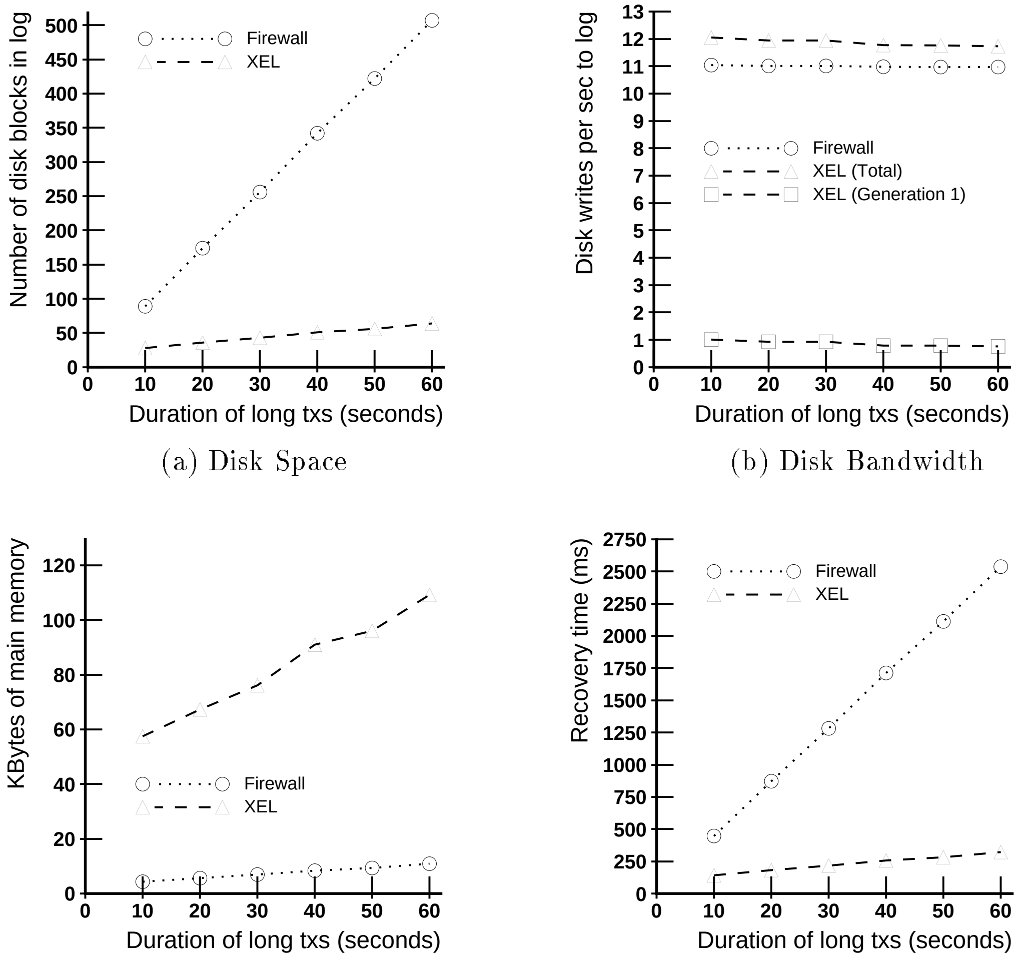
(d) Recovery Time

Figure 6.2: Performance Results for Varying Transaction Mix

0, XEL manages to use 44% less disk space than FW.

6.2.2 Effect of Transaction Duration

Figures 6.3(a)-(d) show the results as the duration of transactions of the long type increases from 10.0 s to 60.0 s in increments of 10.0 s.



(a) Disk Space

(b) Disk Bandwidth

(c) Main Memory

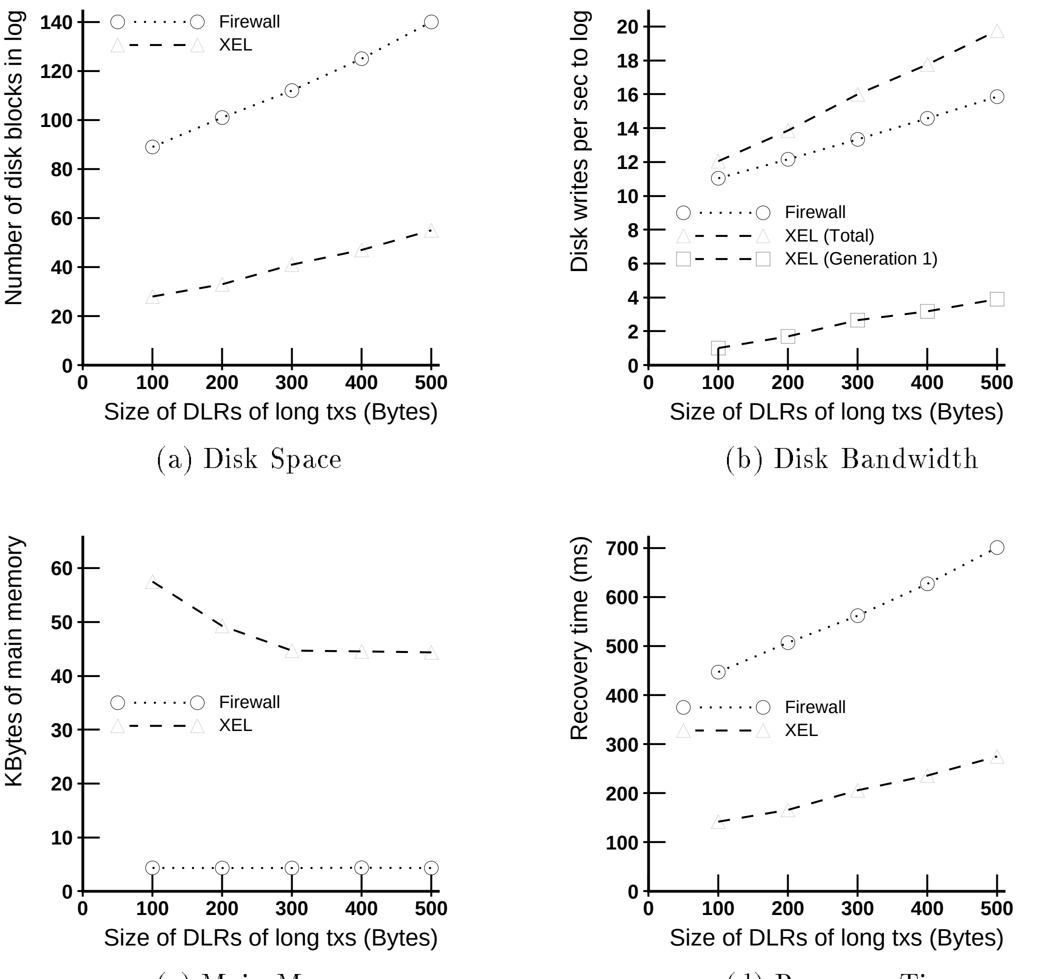
(d) Recovery Time

Figure 6.3: Performance Results for Varying Long Transaction Duration

XEL's advantage over FW increases as the duration of the long transaction type lengthens. For a 60.0 s duration, XEL reduces the size of the log by a factor of 7.9 with only a 6.9% increase in disk bandwidth. Regardless of transaction duration, the average rate of arrival of log records (in steady state) is the same for both FW and XEL. At any given moment, FW must retain all log records that have been written since the first record of the oldest transaction, so the disk space required for FW is roughly proportional to the duration of the longest transaction. However, XEL is able to filter out most log records from short transactions at the head of generation 0, so XEL is largely unaffected by the duration of a small fraction of long transactions.

6.2.3 Effect of Size of Data Log Records

This set of experiments examines the effect of the size of DLRs from the long (10.0 s) transaction type. Each long-lived transaction still writes 4 DLRs, as before, but now the size of each long transaction's DLRs varies from 100 Bytes up to 500 Bytes in increments of 100 Bytes. Figures 6.4(a)-(d) present the results.



(a) Disk Space

(b) Disk Bandwidth

(c) Main Memory

(d) Recovery Time

Figure 6.4: Performance Results for Varying Size of DLRs from Long Transaction Type

As the size of DLRs from long-lived transactions increases, XEL suffers more than FW. The proportion of log information, measured in Bytes, that must be forwarded to generation 1 increases with the size of DLRs from long transactions, so disk space and bandwidth for generation 1 both increase. Furthermore, the bandwidth for generation 0 increases, so records tend to move from tail to head faster. Most records from short transactions are thrown away at the head of generation 0, so their faster movement through generation 0 means that the LM does not need to keep track of them for as long a period of time. This tends to decrease the overall main memory requirements for XEL.

6.2.4 Effect of Data Skew

This section examines the effect of data skew on the performance of XEL for a single log stream. Suppose that there is some subset H of the set of all objects and that the members of H receive a disproportionately large number of updates; these objects are "hot spot" objects because they are updated much more frequently than other objects. Let x, $0 \leq x \leq 1$, be the ratio of the size of H to the total number of objects. Suppose further that when a transaction must choose an object to update, the probability that it chooses a member of H is $1-x$. This simple definition provides a single parameter, x, which represents an application's data skew. By varying x appropriately, it is possible to control the amount of skew in the pattern of updates to data by an application's transactions.

In this section's experiments, x ranges from 5×10^{-5} up to 0.5. In the case of $x=5\times10^{-5}$, almost all the updates affect a set of only 500 objects. Such extremely skewed distributions characterize databases with "hot spot" objects. When $x=0.5$, all objects are updated equally often, on average. Figures 6.5(a)-(d) present the results.

Data skew has only a minor effect on XEL. Even for the most highly skewed distribution, XEL requires only 48% more disk space and 5.5% more bandwidth than it does for a completely unskewed distribution.

6.3 Parallel Logging

This section compares the performances of the three distribution policies for three different data skew specifications as the number of log streams increases. All experiments in this section use the XEL method for disk space management within each log stream.

Let l be the number of log streams. The experiments in this section examine $l=2^k$ for $0 \leq k \leq 6$.

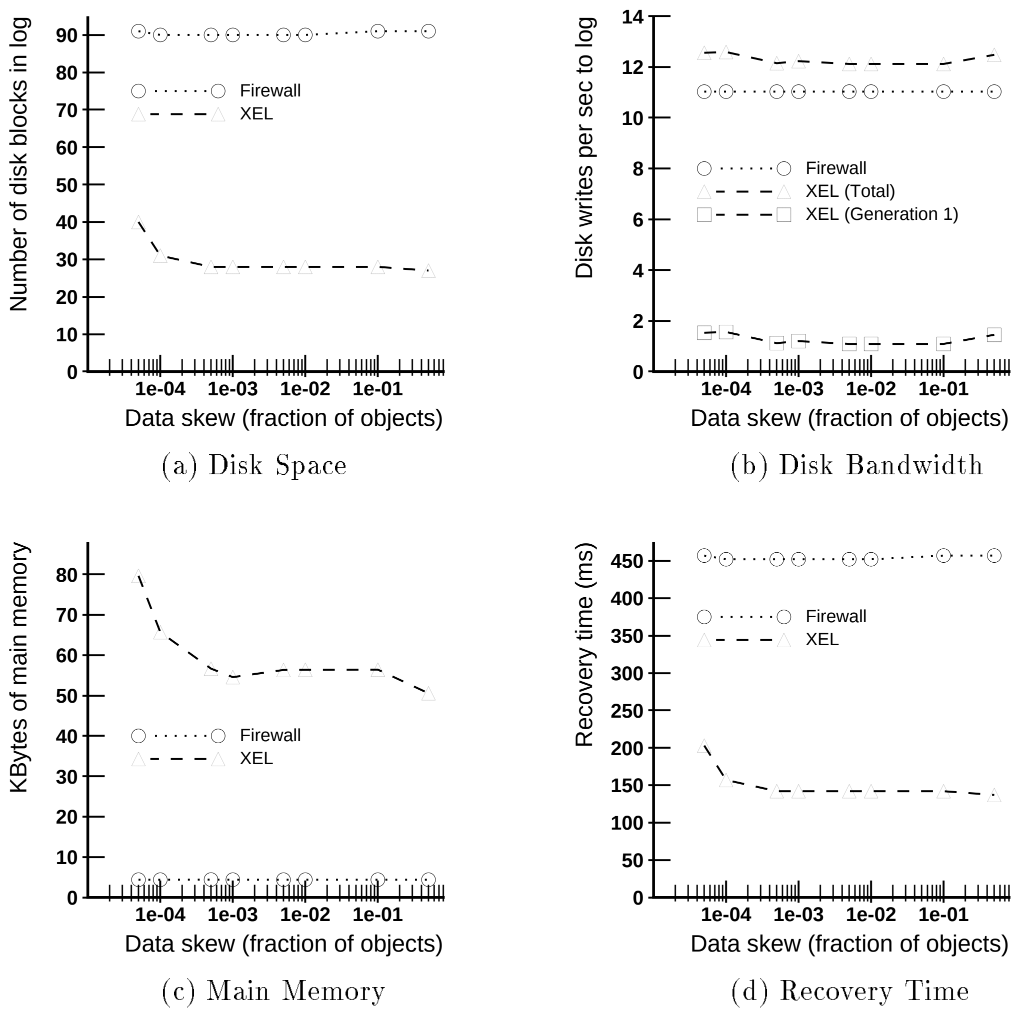


Figure 6.5: Performance Results for Varying Data Skew

Refer to section 6.2.4 for a definition of the data skew parameter, x. The experiments in this section consider three cases of data skew: x=0.5 (no skew), x=0.01 (moderate skew) and $x=2\times10^{-4}$ (high skew). In the case of $x=2\times10^{-4}$, 99.98% of the updates, on average, affect a set of only 2,000 objects.

For the case of maximum data skew and the greatest number of log streams, the average time between consecutive updates for a hot spot object becomes so short that the transaction types defined for the previous sets of experiments (which involved only a single log stream) are no longer feasbile. It is necessary to define new transaction types which would hold their write locks on objects for much shorter durations.

All experiments in this section specify the following two types of transactions. The first is of 0.1 s duration and writes 2 DLRs, each of size 250 Bytes. The second lasts 2.0 s, in which time it writes 2 DLRs of size 250 Bytes each. Their relative probabilities of occurrence are 0.99 and 0.01, respectively.

To provide sufficient bandwidth for flushing updates, each experiment specifies $10\times l$ disk drives with a transfer time of 25 ms each. The conservative 25 ms time allows for some read operations to be interspersed with writes.

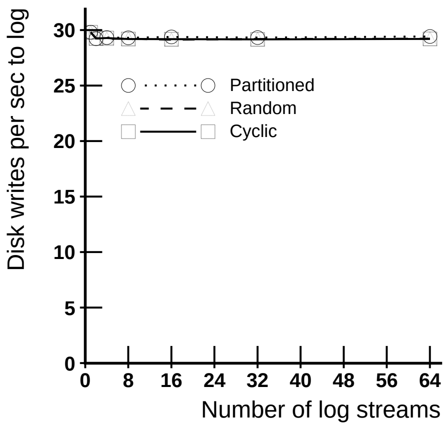
The arrival rate is $100\times l$ TPS and the simulation time is 500 s. All tests use two generations. Recirculation is enabled for $l\geq2$ so that race conditions will not stall any of the log streams [6]. For each skew setting, the sizes of the two generations are found such that disk space is minimized for l=1 (with recirculation disabled), subject to the constraint that no transaction is killed. For $l\geq2$ and x=0.5 or x=0.01, the sizes of generations 0 and 1 are both doubled. For $l\geq2$ and $x=2\times10^{-4}$, the size of generation 0 is quadrupled and double the size of generation 1 is doubled; the larger increase in generation 0 for $x=2\times10^{-4}$ was chosen because it was found to yield substantially lower bandwidth requirements for generation 1.

The evaluation criteria are disk bandwidth and main memory for normal logging activity and elapsed time for recovery. When considering bandwidth, the results reflect the maximum total bandwidth (both generations) for any particular stream.

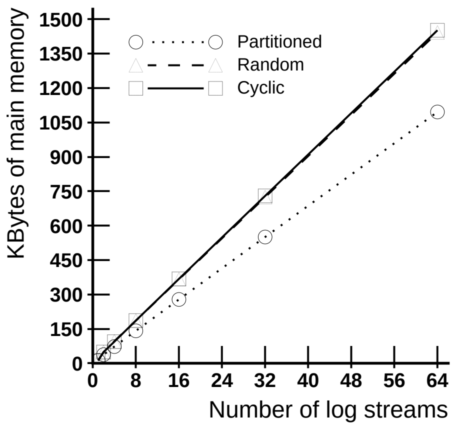
[6] To appreciate the subtlety of potential race conditions, suppose that recirculation in the last generation is disabled and imagine that two different transactions update the same object in quick succession and write DLRs to different streams. Until the first REDO DLR becomes *non-recoverable*, the second DLR must be either *unflushed* or *required* (assuming that no other transaction subsequently updates the object and commits). If the second DLR's stream is "faster" than the stream of the first DLR, it may reach the head of the last generation of its log stream before the first DLR gets to the end of its stream. The fast stream will be forced to stall until the slow stream has caught up. Similar situations can also be imagined (such as for a DLR and the `COMMIT` TLR on which it depends). These interstream dependencies can cause "fast" streams to synchronize with "slow" streams, which is generally undesirable for performance reasons.

6.3.1 No Skew

The results for the case of no skew (x=0.5) are presented in Figure 6.6. The sizes of generations 0 and 1 are 10 and 5 blocks, respectively, for l=1; they are 20 and 10 blocks, respectively, for $l \geq 2$. Recovery time is dominated by the delay required to read in the contents of the log from disk. The elapsed time for recovery is 76 ms for l=1 and 151 ms for $l \geq 2$, regardless of the distribution policy.



(a) Maximum Disk Bandwidth



(b) Main Memory

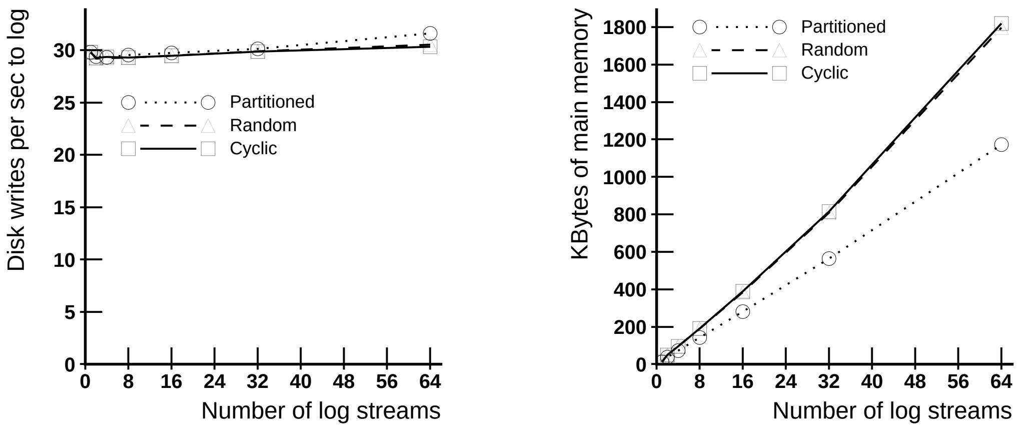
Figure 6.6: Disk Bandwidth and Memory Requirements vs. Parallelism (x=0.5)

All three distribution policies require approximately the same maximum bandwidth, and the maximum bandwidth remains practically constant as the number of log streams increases. Refer to Section 6.3.4 for an explanation of why the *partitioned* distribution policy requires less main memory, compared to the other two policies, for all experiments.

6.3.2 Moderate Skew

Figure 6.7 shows the results for the moderate skew (x=0.01) case. The sizes of generations 0 and 1 are 10 and 5 blocks, respectively, for l=1; they are 20 and 10 blocks, respectively, for $l \geq 2$. The elapsed time for recovery is 76 ms for l=1 and 151 ms for $l \geq 2$, regardless of the distribution policy.

For all three policies, the maximum required bandwidth increases slightly with the number of log streams. This behavior can be attributed to decreasing intervals between successive updates to each object. The transaction arrival rate increases in proportion to the number of log streams but the number of hot spot objects remains constant, so the average duration between successive updates to any particular object is inversely proportional to the number of log streams. At this skew setting, a REDO DLR becomes



(a) Maximum Disk Bandwidth

(b) Main Memory

Figure 6.7: Disk Bandwidth and Memory Requirements vs. Parallelism (x=0.01)

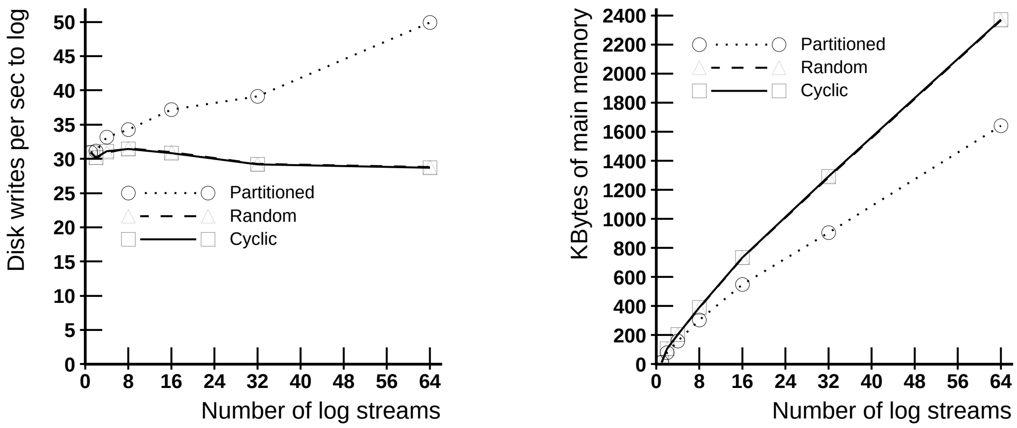
more likely to have a *required* status, because of the lingering presence of a *recoverable* DLR from a prior update to the same object, when the LM must decide whether or not to forward the DLR from generation 0 to generation 1. In terms of maximum bandwidth, the partitioned policy performs slightly worse than the other two because of static skew effects.

The non-linear slope for main memory requirements can be explained similarly. The `COMMIT` records from transactions are more likely to be forwarded into generation 1 because *required* REDO DLRs depend on them. Therefore, `COMMIT` records tend to survive longer and the LM must continue to pay attention to all associated DLRs; previously, the LM would have thrown away many of these `COMMIT` records at the head of generation 0 instead of forwarding them, and any remaining REDO DLRs would instantly become *non-recoverable*.

6.3.3 High Skew

Figure 6.8 shows the results for the case of high skew (x=2×10^{-4}). The sizes of generations 0 and 1 are 10 and 5 blocks, respectively, for l=1; they are 40 and 10 blocks, respectively, for $l\geq2$. The elapsed time for recovery is 76 ms for l=1 and 252 ms for $l\geq2$, regardless of the distribution policy.

It is interesting to note that the bandwidth curves for the *cyclic* and *random* policies slope up and then down, as the number of log streams increases. The initial increases have a similar explanation as was given for the case of moderate skew. Namely, a REDO DLR becomes more likely to be forwarded to generation 1 as the mean time



(a) Maximum Disk Bandwidth

(b) Main Memory

Figure 6.8: Disk Bandwidth and Memory Requirements vs. Parallelism (x=2×10^{-4})

between updates decreases. As the throughput increases even further, the average time between consecutive updates to an object becomes so short that a DLR is more likely to become only *recoverable* by the time the LM must decide whether or not to forward it to generation 1 because another transaction has already updated the same object and committed.

6.3.4 Discussion

Regardless of skew, the *partitioned* distribution policy requires less main memory than either *cyclic* or *random* because it takes advantage of the fact that all DLRs for a particular object are directed to the same log stream; the object's LOT entry can be located at the same processor node as the one that manages the cells for its DLRs and can directly point to these cells. With the *cyclic* and *random* policies, an object's LOT entry no longer points directly to the cells for its DLRs. Instead, it indicates the streams where there are *relevant* DLRs for the object; local LOT tables at those streams point directly to the corresponding cells. This indirection entails higher main memory requirements.

In terms of disk bandwidth, all three distribution policies are approximately the same for low data skew. For high data skew, *partitioned* suffers noticeably from static skew effects and so it requires significantly more disk bandwidth than the other two. The *random* and *cyclic* strategies both exhibit approximately the same behavior for all data skews.

Chapter 7

Conclusions

7.1 Lessons Learned

This thesis has proposed and evaluated a new variation of logging and recovery that is well suited to highly concurrent database systems. Extended ephemeral logging (XEL), a new disk management method that does not require periodic checkpoint operations, is the cornerstone upon which the rest of the logging and recovery system is built. An application's bandwidth requirements for log information may demand a collection of log streams which work in parallel. Each log stream resides on a single disk drive or possibly a small set of drives. XEL manages the disk space within each log stream.

Chapter 3 proved important safety and liveness properties for a simplified version of XEL that was expressed in terms of the I/O automata model [42], thereby imparting confidence in the correctness of XEL.

XEL's performance was experimentally evaluated in Chapter 6, using an event-driven simulator. XEL was compared to the traditional "firewall" (FW) disk management method for a single stream of log records and the experimental results suggest that XEL's advantage over FW increases under any of the following conditions:

- The lifetime of an application's longest transaction type increases.
- The amount of log information written by an application's longest transaction type decreases.
- The probability of occurrence of an application's longest transaction type decreases (but remains non-zero).

When a system has multiple log streams, XEL can accommodate any distribution policy but the *partitioned* policy generally requires less main memory space. Unless an application updates a relatively small collection of objects much more frequently than other objects in the database, all three oblivious distribution policies which Chapter 4 considered yield approximately equal loads on all log streams. However, the *random* and *cyclic* distribution policies lead to better load balancing, compared to the *partitioned* policy, if an application has a small collection of "hot spot" objects.

Log streams need not be especially large, in terms of storage space, when a system's LM uses XEL to manage the disk space allocated for log information. Small log streams and large main memories enable much faster recovery after a crash; a database system's recovery manager (RM) can sequentially retrieve a log stream's contents from disk and process them in a single pass. When multiple log streams exist, the RM processes them all independently in parallel.

7.2 Importance of Results

This thesis widens the options available to DBMS designers. The firewall (FW) method's abstraction of a single FIFO queue for log information is inappropriate in some circumstances. In such circumstances, a DBMS designer may prefer to use the XEL method instead.

If a small fraction of transactions have relatively long lifetimes, XEL retains the necessary log information from these long transactions but reclaims the disk space occupied by log records from much shorter transactions. Therefore, XEL can significantly reduce the size of the log for some applications. The primary benefit of a much smaller log is faster recovery after a crash. A smaller log may also decrease a system's cost.

Checkpoints are no longer a necessity with XEL. This eliminates the overhead (in terms of computation, communication, disk bandwidth and disk space) and complexity that accompany any disk management method which involves checkpoints (e.g., the FW method). This advantage is especially welcome in highly concurrent systems that have an arbitrarily large number of log streams and an arbitrarily large number of disk drives on which the disk version of the database is kept; coordination for periodic checkpoints in such a highly concurrent setting becomes cumbersome.

An arbitrarily large collection of disk drives provide the necessary bandwidth for log information. As the size of this collection grows, the single FIFO queue abstraction becomes increasingly awkward to implement. A more convenient abstraction is to view the log as a collection of *log streams* that operate largely independently of each other. This abstraction can be implemented efficiently if XEL is used to manage the disk space within each log stream.

The simulation results presented in this thesis quantitatively demonstrate XEL's effectiveness for a wide variety of applications and illustrate its strengths and weaknesses compared to the traditional FW method.

7.3 Extensions

7.3.1 Non-volatile Region of Main Memory

Previous authors [15, 13, 39, 9, 53] have proposed system designs in which some (but not all) of main memory is non-volatile. Battery backup to some portions of RAM ensures that the contents will not be lost if the regular power supply is interrupted. In such a system, parallel XEL can greatly reduce the disk bandwidth required for log information so that much fewer disk drives are needed for the log. The youngest generation (generation 0) for each log stream can be kept in non-volatile main memory; the LM writes log records to disk only when space in non-volatile main memory has been exhausted.

7.3.2 Log-Only Disk Management

Suppose a computer's main memory provides sufficient capacity to hold all the objects of a database and that applications update most of these objects quite frequently. In such a setting, a separate disk version of the database is superfluous. The most recently committed value for each object can be kept in only the log. A few small changes to the XEL algorithm yield a variant that meets the needs of this log-only situation. Without a disk version of the database, UNDO DLRs are no longer needed. The status of each REDO DLR is either *required* or *not-required*; a REDO DLR has status *required* if the transaction which performed the update is still in progress or if the DLR is for the most recently committed update to the object. Although older REDO DLRs for the same object may still be recoverable, the LM doesn't need to keep track of them because it will never erase the DLR for the most recently committed update to the object; hence, these older REDO DLRs all have status *not-required.* Likewise, a `COMMIT` record can have a status of either *required* or *not-required*; a `COMMIT` record is *required* if any only if at least one DLR from the transaction is still *required.* The LM keeps track of only `COMMIT` records which are *required.*

This log-only variant of XEL offers several advantages. First, it eliminates the expense and complexity that arise from managing a separate disk version of the database. Second, the LM no longer needs to keep track of old stale records in the log, and this implies lower main memory storage requirements for the LOT and LTT. Furthermore, the fact that the LM keeps track of only *required* records means that the status field can be eliminated from each cell, thus yielding further reductions in main memory requirements.

7.3.3 Multiplexing of Log Streams

One possible drawback to XEL is that it may cause write operations to non-sequential positions on disk. Movement between generations within a stream introduces random disk I/O, which is generally much less efficient than sequential I/O. For example, suppose a log stream has two generations. When the LM must occasionally write a block of forwarded log records to the tail of generation 1, it must seek to this track's location on disk and wait for the block's location to rotate under the disk head. When the write to generation 1 has finished, the LM returns to the tail of generation 0, where it can resume writing blocks to generation 0 in sequential order.

A LM with a sufficiently large collection of log disk drives can alleviate the need for occasional non-sequential accesses to the log by multiplexing older generations from different streams, as illustrated in Figure 7.1 for a LM whose streams have two generations.

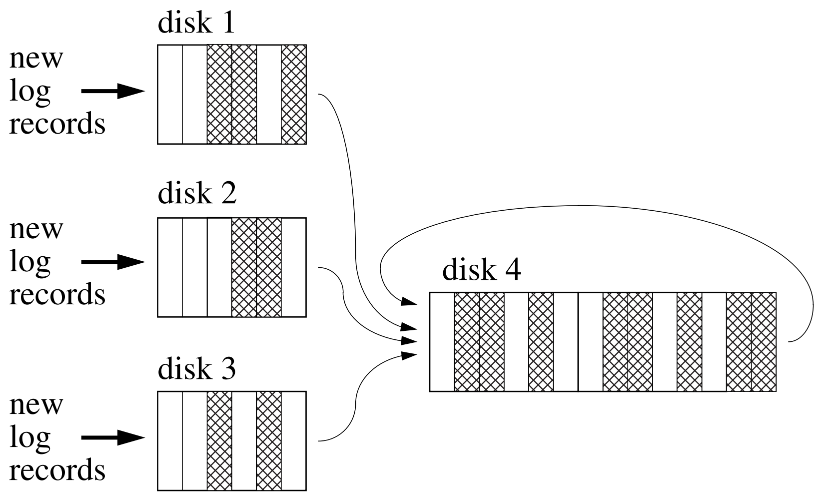


Figure 7.1: Multiplexing of Older Generations

With this configuration, the LM can exploit completely sequential disk I/O when writing log information to generation 0 of any particular log stream. When the LM must forward log records to generation 1, it writes them to a physically different disk drive (or set of drives) so that no random I/O is required.

This technique of multiplexing older generations from different streams permits quick reclamation of the disk space allocated to log records from short transactions but doesn't suffer from any performance degradations due to random disk I/O to the log disks.

7.3.4 Choice of Generation Sizes

For a particular application, how many generations should each log stream have, and what should be the sizes of these generations? Currently, no analytical methods are available to answer these questions. The experiments reported in Chapter 6 relied on simulation to determine the optimal configuration for each particular case. Therefore, formulation of an accurate analytic model to determine the optimal number of generations and their sizes for any particular application remains a challenging open problem.

The characteristics of an application may vary over time, and so the design of an enhanced version of XEL that can adaptively alter its parameters in response to changing conditions is another important open problem.

7.3.5 Fault Tolerance to Isolated Media Failures

This thesis has concentrated on fault tolerance to *system failures* (crashes) in which the contents of main memory are lost but all information on non-volatile disk storage remains intact. To tolerate *media failures*, in which information on disk is lost, a DBMS must exploit redundancy. RAID [52, 51] and Parity Striping [23] have both recently been proposed as solutions to the problem of isolated media failures. A parallel implementation of XEL and RAID can be combined with little difficulty. Disk I/O for log information is characterized by sequential transfers of large blocks of information, and so level 3 RAID is most appropriate. A group of disk drives, one of which is used only for parity information, constitute each log stream. The maximum bandwidth per stream is now higher (compared to the simplistic situation of only one disk drive per log stream), so fewer streams are required. In contrast, I/O for the disk version of the database is characterized by random requests for small pieces of data, so level 5 RAID or Parity Striping would be the best choice for it. The differing requirements of the LM and CM provide a good example of a situation in which it is advantageous to employ two different levels of RAID in the same system.

RAID systems can be designed to provide very high reliability. For example, a 1000 disk level 5 RAID with a group size of 10 and a few standby spares could have a calculated MTTF (mean time to failure) of over 45 years [52]. Isolated media failures pose very little threat under such circumstances. Other components of the system may likely fail before an irrecoverable media failure happens. Nevertheless, the mere threat of an irrecoverable media failure may warrant attempts for further fault tolerance. A familiar solution (see [5], for example) is to periodically dump the database's current state to an archive version and maintain an archive log of all transactions which have executed since the archive version was dumped. The archive version of the database and the archive log may be kept on tape rather than on disk. This thesis has not addressed the problem of maintaining an archive log for a database which requires very high bandwidth for log information. Another option, which is already practiced by some large commercial users of databases, is to have two duplicate database systems running at geographically separate sites to that a natural or man-made disaster at one site does not wipe out all data. This option may be expensive, because it requires full duplication, but it does provide good fault tolerance. In the event of a media failure at one site, the other site still offers an up-to-date version of the database. The likelihood of simultaneous failures at both sites ought to be very low. The provision of more efficient means to ensure fault tolerance to irrecoverable media failures remains a challenging open problem.

7.3.6 Fault Tolerance to Partial Failures

This thesis has conveniently assumed that the concurrent system on which the DBMS executes is either completely up or completely down. In practice, some machines may be able to continue operation at some processing nodes despite partial failures at other nodes elsewhere within the same machine. When the system hardware allows such "graceful degradataion", a DBMS designer might wish to structure the DBMS software so that it too allows graceful degradation.

One obvious way to provide fault tolerance to partial failures is to duplicate all an application's data structures and code. For example, all LOT entries at a particular node would be duplicated at some other node elsewhere in the machine. Any update to one copy of the these LOT entries must be applied to the other copy as well. Refer to [4] for an example of a system which exploits software redundancy to tolerate partial failures.

7.3.7 Support for Transition Logging

This thesis has always assumed physical state logging at the access path level. Some DBMS designers may prefer other styles of logging. For example, the ARIES method for logging and recovery [45] uses transition logging [30] and incorporates *compensation log records (CLRs)* to undo the effects of previous updates to objects. If a LM performs transition logging for an object and a transaction updates the object, the resulting log record indicates only the operation that transformed the object's old value into its new value; neither the old state nor the new state of the object is represented in the log. In this context, what is the definition of a *garbage* log record, and what is a *relevant* log record? How should the LM manage the LOT and LTT?

When a transaction performs an operation on an object, the resulting log record shall be called a FWD record (since it refers to an operation in the "forward" direction in time). When the LM undos an operation, the resulting record is a CLR, as mentioned above. In general, an operation described in a log record is not idempotent. If the RM must undo an update by a transaction, it must first be sure that the version of the object in the disk version of the database actually incorporates the transaction's original update, lest an unwarranted undo action put the object into an incorrect state. One way to synchronize the log and the disk version of the database is to keep a timestamp with every object upon which the LM will perform transition logging. Suppose this timestamp is an integer-valued counter (initially 0) and the LM increments the counter every time it changes the object. Whenever the CM flushes an updated object to the disk version of the database, the object's current timestamp accompanies it and resides with it in the disk version of the database. Whenever the LM modifies an object (in either the "forward" or "backward" direction), it increments the timestamp and stores the new timestamp value in the associated log record. When the RM must restore an object to its most recently committed pre-crash state, it knows that the version of the object in the disk version of the database already incorporates the effects of all operations whose log records have timestamps less than or equal to that found with the object in the disk version of the database. The RM must redo only those operations which are described in subsequent log records from committed transactions. Similarly, the RM must undo any operations that were performed by transactions which aborted or were interrupted by the crash; some of these operations may temporally precede the current version of the object in the disk version of the database.

The LM must retain all FWD and CLR log records for operations that temporally follow the current version of the object in the disk version of the database. It must also retain all FWD log records from uncommitted transactions that do not have corresponding CLRs, regardless of whether these records temporally precede or follow the current version of the object in the disk version of the database. If a transaction commits, the LM must retain its `COMMIT` record until all FWD log records from the transaction have been overwritten; otherwise, the RM could find a FWD log record which appears to be from an uncommitted transaction but the log contains no subsequent CLR so the RM ought to undo the operation described in the FWD record. If the LM writes a FWD log record and later writes a CLR to undo the operation, it must retain the CLR in the log until the FWD record has been overwritten; this ensures that the RM does not undo an operation which has already been undone. These considerations dictate when `COMMIT`, FWD and CLR records become garbage. The LOT and LTT can track the positions and status values for FWD and CLR records just as it did for other types of log records.

7.3.8 Adaptive Distribution Policies

This thesis demonstrated that a simple oblivious distribution policy, such as *random* or *cyclic*, can ensure excellent load balancing behavior in a system with arbitrarily many log streams, where load balancing is defined in terms of the demanded bandwidth of each stream. However, there may still be reasons to pursue more sophisticated distribution policies.

An adaptive distribution policy takes into account the current state of the system when deciding the stream to which to send a log record; any such strategy is clearly not oblivious. Although an adaptive strategy cannot achieve any improvements in terms of balancing the demanded bandwidths of log streams, it can offer other benefits. Each log stream has a fixed number of buffers. If one stream's buffers are all completely full temporarily, an adaptive strategy ought to redirect log records to other streams which still have buffer capacity available. Furthermore, the *random* and *cyclic* policies make no attempt to exploit locality within a concurrent computer. As a system's degree of concurrency scales up, global network bandwidth may become more limited and locality may affect a database's overall performance. The *partitioned* policy, despite its drawbacks in terms of load balancing, can provide good locality; a modified variant of the *partitioned* policy may yield the best solution, in terms of both load balancing and locality. Log records are sent to streams according to the *partitioned* policy. However, an overloaded log stream does not refuse to accept records if it can redirect them to some other stream (preferably one that is nearby) that can accept more records.

Theoretical analysis of an adaptive distribution policy becomes quite complicated because the system incorporates feedback. Such problems have classically been the preserve of control systems theory. Theoretical analysis and experimental evaluation of an adapative policy must take into the system's dynamic behavior. For example, a system might exhibit oscillatory behavior under some conditions. These issues ought to be thoroughly understood before any adaptive policy is chosen for use within a database system.

7.3.9 Quick Resumption of Service After a Crash

The description of the RM's operation in Section 2.9 stated that normal operation resumes after recovery activity has completely finished. For some applications, availability is very important and normal activity should resume as quickly as possible after a crash. More sophisticated recovery algorithms for XEL may permit a database to start servicing requests before recovery has entirely completed; normal operation and recovery are overlapped. This remains an interesting open problem.

7.4 The Future of High Performance Databases

In the future, information will be plentiful, available and inexpensive; but it won't be free. As computer and communications technology becomes pervasive in our society, many offices and homes will be able to retrieve information from large databases. In general, the corporations which provide these information services will charge the consumers appropriately. Fees will reflect the value of the information to the consumer and the cost to the producer for gathering, storing and distributing the information. Database technology will provide not only the ability to store and retrieve large amounts of diverse information. It will also provide a means to charge customers accordingly, so as to ensure economic efficiency.

Suppose an "information vendor" establishes a database which provides information services to many millions of customers. If 10,000,000 customers happen to be using the system at one time and each customer submits approximately one request every 100 seconds, say, this generates a load of 100,000 TPS (transactions per second) for the billing database. This is enormous, compared to the best demonstrated performance of today's systems.

The cost for maintaining billing information is important. It ought to be relatively low, compared to the cost of the information itself, so that the price is not significantly inflated by the need to charge a fee for each request.

These considerations suggest that high performance, low cost transaction processing systems will play an important role in our society as we become information consumers and many hundreds of millions of people and companies buy and sell information.

The work in this thesis is a step toward realizing this dream. However, it addresses only a necessary condition (fault tolerance), not a sufficient condition. Much more work remains to be done in many other areas. The problems of concurrency control, query optimization and transaction management all deserve re-examination within the context of highly concurrent database systems that support very high rates of transaction processing.

Appendix A

Theorems for Correctness Proof of XEL

This appendix contains proofs for the safety and liveness properties of XEL. These proofs supplements Chapter 3 in the main body of the thesis.

A.1 Proof of Possibilities Mapping

The following lemmas and theorems will prove that f, as defined in Section 3.4.3, satisfies the sufficient conditions (stated in [42, 43]) to be a possibilities mapping from `LM` to `SLM`.

Theorem A.1 $\forall \mathrm{s}_0,\ \mathrm{s}_0 \in \mathrm{start}(\texttt{LM}),\ \exists \mathrm{t}_0 \text{ s.t. } (\mathrm{t}_0 \in \mathrm{start}(\texttt{SLM})) \wedge (\mathrm{t}_0 \in \mathrm{f}(\mathrm{s}_0))$

Proof:

- $$\begin{array}{ll} ((\ & (\text{curr\_reqd\_dlr}=\perp) \\ \wedge & (\text{pending\_ts\_assign}=\emptyset) \\ \wedge & (\forall x,\ x\in\mathcal{N},\ \text{timestamp}_x=\perp) \\ \wedge & (\forall x,\ x\in\mathcal{N},\ \text{status}_x=\text{UNFL})\ ) \quad \text{in state } \mathrm{s}_0) \end{array}$$
  $\wedge\ (\mathrm{t}\in\mathrm{f}(\mathrm{s}_0))$
  $\Longrightarrow ((\text{keep}=\perp) \wedge (\text{let\_erase}=\emptyset) \wedge (\text{wait\_erase}=\emptyset))$ in state t
- start(`SLM`)=$\{\mathrm{t}_0\}$
  where $((\text{keep}=\perp) \wedge (\text{let\_erase}=\emptyset) \wedge (\text{wait\_erase}=\emptyset))$ in state t_0
- $((\text{keep}=\perp) \wedge (\text{let\_erase}=\emptyset) \wedge (\text{wait\_erase}=\emptyset))$ in state $\mathrm{t}_0 \Longrightarrow \mathrm{t}=\mathrm{t}_0$
- $(\mathrm{t}=\mathrm{t}_0) \wedge (\mathrm{t}\in\mathrm{f}(\mathrm{s}_0)) \Longrightarrow \mathrm{t}_0\in\mathrm{f}(\mathrm{s}_0)$
  and thus the theorem has been proven. □

Lemma A.2

$$
\begin{array}{ll}
 & (\text{lm\_keep}(x) \text{ in state s}) \\
\wedge & (\text{t}\in\text{f(s)}) \\
\Longrightarrow & \\
 & \text{keep}=x \text{ in state t}
\end{array}
$$

Proof:

By contradiction. Assume keep$\neq x$ in state t.

- (t$\in$f(s)) $\wedge$ (keep$\neq x$ in state t)
  $\Longrightarrow$ Either
  - (1) (lm_let(x) in state s) $\wedge$ ($x\in$let_erase in state t)
    - lm_let(x) in state s $\Longrightarrow$ $\neg$lm_keep(x) in state s
      by definition of lm_let(x) and lm_keep(x)

    But this is a contradiction, so this case cannot be true.

  or
  - (2) (lm_wait(x) in state s) $\wedge$ ($x\in$wait_erase in state t)
    - lm_wait(x) in state s
      $\Longrightarrow$ status$_x$=RECV in state s — by definition of lm_wait(x)
    - status$_x$=RECV in state s
      $\Longrightarrow$ curr_reqd_dlr$\neq x$ in state s — by Invariant 3.6
    - curr_reqd_dlr$\neq x$ in state s
      $\Longrightarrow$ $\neg$lm_keep(x) in state s — by definition of lm_keep(x)

    But this is a contradiction, so this case cannot be true.

  or
  - (3) ($\neg$recvbl(x) in state s)
    $\wedge$ (((keep$\neq x$) $\wedge$ ($x\notin$let_erase) $\wedge$ ($x\notin$wait_erase)) in state t)
    - lm_keep(x) in state s
      $\Longrightarrow$ curr_reqd_dlr=x in state s — by definition of lm_keep(x)
    - curr_reqd_dlr=x in state s $\Longrightarrow$ recvbl(x) in state s
      by Invariants 3.6 and 3.8 and definition of recvbl(x)

    But this is a contradiction, so this case cannot be true.
- Therefore, every possible case leads to a contradiction and so the original assumption must be false. Thus the lemma has been proven. □

Lemma A.3

$$
\begin{array}{ll}
 & (\text{lm\_let}(x) \text{ in state s}) \\
\wedge & (\text{t}\in\text{f(s)}) \\
\Longrightarrow & \\
 & x\in\text{let\_erase in state t}
\end{array}
$$

Proof:

By contradiction. Assume $x\notin$let_erase in state t.

- (t$\in$f(s)) $\wedge$ ($x\notin$let_erase in state t)
  $\Longrightarrow$ Either

(1) (lm_keep(x) in state s) $\wedge$ (keep=x in state t)

- lm_keep(x) in state s $\Longrightarrow$ $\neg$lm_let(x) in state s
  by definition of lm_keep(x) and lm_let(x)

  But this is a contradiction, so this case cannot be true.

or

(2) (lm_wait(x) in state s) $\wedge$ ($x\in$wait_erase in state t)

- lm_wait(x) in state s
  $\Longrightarrow$ status$_x$=RECV in state s by definition of lm_wait(x)
- status$_x$=RECV in state s
  $\Longrightarrow$ curr_reqd_dlr$\neq x$ in state s by Invariant 3.6
- ((curr_reqd_dlr$\neq x$) $\wedge$ (lm_wait(x))) in state s
  $\Longrightarrow$ $\neg$lm_let(x) in state s by definition of lm_let(x)

  But this is a contradiction, so this case cannot be true.

or

(3) ($\neg$recvbl(x) in state s)
$\wedge$ (((keep$\neq x$) $\wedge$ ($x\notin$let_erase) $\wedge$ ($x\notin$wait_erase)) in state t)

- lm_let(x) in state s
  $\Longrightarrow$ ((curr_reqd_dlr=x) $\vee$ (recvbl(x))) in state s
  by definition of lm_let(x)
- ((curr_reqd_dlr=x) $\vee$ (recvbl(x))) in state s
  $\Longrightarrow$ recvbl(x) in state s
  by Invariants 3.6 and 3.8 and definition of recvbl(x)

  But this is a contradiction, so this case cannot be true.

- Therefore, every possible case leads to a contradiction and so the original assumption must be false. Thus the lemma has been proven. □

Lemma A.4
(lm_wait(x) in state s)
$\wedge$ (t$\in$f(s))
$\Longrightarrow$
$x\in$wait_erase in state t

Proof:

By contradiction. Assume $x\notin$wait_erase in state t.

- (t$\in$f(s)) $\wedge$ ($x\notin$wait_erase in state t)
  $\Longrightarrow$ Either

  (1) (lm_keep(x) in state s) $\wedge$ (keep=x in state t)

  - lm_keep(x) in state s
    $\Longrightarrow$ curr_reqd_dlr=x in state s by definition of lm_keep(x)
  - curr_reqd_dlr=x in state s
    $\Longrightarrow$ status$_x\neq$RECV in state s by Invariant 3.6
  - status$_x\neq$RECV in state s
    $\Longrightarrow$ $\neg$lm_wait(x) in state s by definition of lm_wait(x)

    But this is a contradiction, so this case cannot be true.

or

(2) (lm_let(x) in state s) $\wedge$ ($x \in$let_erase in state t)

- lm_let(x) in state s
  $\Longrightarrow$ ((curr_reqd_dlr=x) $\vee$ ($\neg$lm_wait(x))) in state s
  by definition of lm_let(x)
- ((curr_reqd_dlr=x) $\vee$ ($\neg$lm_wait(x))) in state s
  $\Longrightarrow$ $\neg$lm_wait(x) in state s by Invariant 3.6

  But this is a contradiction, so this case cannot be true.

or

(3) ($\neg$recvbl(x) in state s)
$\wedge$ (((keep$\neq x$) $\wedge$ ($x \notin$let_erase) $\wedge$ ($x \notin$wait_erase)) in state t)

- lm_wait(x) in state s
  $\Longrightarrow$ status$_x$=RECV in state s by definition of lm_wait(x)
- status$_x$=RECV in state s
  $\Longrightarrow$ timestamp$_x \in \mathcal{N}$ in state s by Invariant 3.7
- ((status$_x$=RECV) $\wedge$ (timestamp$_x \in \mathcal{N}$)) in state s
  $\Longrightarrow$ recvbl(x) in state s by definition of recvbl(x)

  But this is a contradiction, so this case cannot be true.

- Therefore, every possible case leads to a contradiction and so the original assumption must be false. Thus the lemma has been proven. □

Lemma A.5 ($\neg$recvbl(x) in state s)
$\wedge$ (t$\in$f(s))
$\Longrightarrow$
((keep$\neq x$) $\wedge$ ($x \notin$let_erase) $\wedge$ ($x \in$wait_erase)) in state t

Proof:

By contradiction.

Assume ((keep=x) $\vee$ ($x \in$let_erase) $\vee$ ($x \in$wait_erase)) in state t.

- (t$\in$f(s)) $\wedge$ (((keep=x) $\vee$ ($x \in$let_erase) $\vee$ ($x \in$wait_erase)) in state t)
  $\Longrightarrow$ Either

(1) (lm_keep(x) in state s) $\wedge$ (keep=x in state t)

- lm_keep(x) in state s
  $\Longrightarrow$ curr_reqd_dlr=x in state s by definition of lm_keep(x)
- curr_reqd_dlr=x in state s $\Longrightarrow$ recvbl(x) in state s
  by Invariants 3.6 and 3.8 and definition of recvbl(x)

  But this is a contradiction, so this case cannot be true.

or

(2) (lm_let(x) in state s) $\wedge$ ($x \in$let_erase in state t)

- lm_let(x) in state s
  $\Longrightarrow$ ((curr_reqd_dlr=x) $\vee$ (recvbl(x))) in state s
  by definition of lm_let(x)

- ((curr_reqd_dlr=x) $\vee$ (recvbl(x))) in state s
  $\Longrightarrow$ recvbl(x) in state s
  by Invariants 3.6 and 3.8 and definition of recvbl(x)

  But this is a contradiction, so this case cannot be true.

or

(3) (lm_wait(x) in state s) $\wedge$ ($x$$\in$wait_erase in state t)

- lm_wait(x) in state s $\Longrightarrow$ status$_x$=RECV in state s
  by definition of lm_wait(x)
- status$_x$=RECV in state s
  $\Longrightarrow$ timestamp$_x$$\in\mathcal{N}$ in state s by Invariant 3.7
- ((status$_x$=RECV) $\wedge$ (timestamp$_x$$\in\mathcal{N}$)) in state s
  $\Longrightarrow$ recvbl(x) in state s by definition of recvbl(x)

  But this is a contradiction, so this case cannot be true.

- Therefore, every possible case leads to a contradiction and so the original assumption must be false. Thus the lemma has been proven. □

Lemma A.6 $\wedge$ (recvbl(x), $x\in\mathcal{N}$, in state s)
$\wedge$ (t$\in$f(s))
$\Longrightarrow$
((keep=x) $\vee$ ($x$$\in$let_erase) $\vee$ ($x$$\in$wait_erase)) in state t

Proof:

By contradiction.
Assume ((keep$\neq x$) $\wedge$ ($x$$\notin$let_erase) $\wedge$ ($x$$\notin$wait_erase)) in state t.

- (t$\in$f(s)) $\wedge$ (((keep$\neq x$) $\wedge$ ($x$$\notin$let_erase) $\wedge$ ($x$$\notin$wait_erase)) in state t)
  $\Longrightarrow$ $\neg$recvbl(x) in state s by Definition 3.1

  But this contradicts the lemma's predicate. Therefore, the initial assumption must be false and thus the lemma has been proven. □

Lemma A.7 (s is a reachable state of `LM`)
$\wedge$ (recvbl(x), $x\in\mathcal{N}$, in state s)
$\wedge$ (t$\in$f(s))
$\Longrightarrow$
((let_erase$\neq\emptyset$) $\vee$ (wait_erase$\neq\emptyset$)) in state t

Proof:

- (recvbl(x) in state s) $\wedge$ (t$\in$f(s))
  $\Longrightarrow$ ((keep=x) $\vee$ ($x$$\in$let_erase) $\vee$ ($x$$\in$wait_erase)) in state t
  by Lemma A.6
- Either

  (1) keep=x in state t

- t is a reachable state
  $\Longrightarrow$ ((keep=$\perp$) $\vee$ (let_erase$\neq\emptyset$) $\vee$ (wait_erase$\neq\emptyset$)) in state t
  by Invariant 3.1
- (keep=x, $x\in\mathcal{N}$, in state t)
  $\wedge$ (((keep=$\perp$) $\vee$ (let_erase$\neq\emptyset$) $\vee$ (wait_erase$\neq\emptyset$)) in state t)
  $\Longrightarrow$ ((let_erase$\neq\emptyset$) $\vee$ (wait_erase$\neq\emptyset$)) in state t

or

(2) keep$\neq x$ in state t

- (keep$\neq x$, $x\in\mathcal{N}$, in state t)
  $\wedge$ (((keep=x) $\vee$ ($x\in$let_erase) $\vee$ ($x\in$wait_erase)) in state t)
  $\Longrightarrow$ (($x\in$let_erase) $\vee$ ($x\in$wait_erase)) in state t
- (($x\in$let_erase) $\vee$ ($x\in$wait_erase)) in state t
  $\Longrightarrow$ ((let_erase$\neq\emptyset$) $\vee$ (wait_erase$\neq\emptyset$)) in state t

Therefore, the desired result is obtained for both possible cases and thus the lemma has been proven. □

Lemma A.8 (s is a reachable state of LM)
$\wedge$ ($\nexists x$, $x\in\mathcal{N}$, s.t. recvbl(x), in state s)
$\wedge$ (t$\in$f(s))
$\Longrightarrow$
((let_erase=$\emptyset$) $\wedge$ (wait_erase=$\emptyset$)) in state t

Proof:

By contradiction. Assume ((let_erase$\neq\emptyset$) $\vee$ (wait_erase$\neq\emptyset$)) in state t

- (((let_erase$\neq\emptyset$) $\vee$ (wait_erase$\neq\emptyset$)) in state t)
  $\wedge$ ((($\perp\notin$let_erase) $\wedge$ ($\perp\notin$wait_erase)) in state t)
  $\Longrightarrow$ $\exists x$, $x\in\mathcal{N}$, s.t. (($x\in$let_erase) $\vee$ ($x\in$wait_erase)) in state t
  by Invariant 3.5
- ((($x\in$let_erase) $\vee$ ($x\in$wait_erase)) in state t) $\wedge$ (t$\in$f(s))
  $\Longrightarrow$ ((lm_let(x)) $\vee$ (lm_wait(x))) in state s by Lemmas A.3 and A.4
- ((lm_let(x)) $\vee$ (lm_wait(x))) in state s $\Longrightarrow$ recvbl(x) in state s
  by definitions of lm_let(x) and lm_wait(x),
  and by Invariants 3.6, 3.7 and 3.8

But this contradicts the lemma's predicate, and so the original assumption must be false. Thus the lemma has been proven. □

Theorem A.9 (s_{i-1} is a reachable state of LM)
$\wedge$ (π_i=$\mathtt{COMMIT}_x$)
$\wedge$ ((s_{i-1},$\mathtt{COMMIT}_x$,s_i)$\in$steps(LM))
$\wedge$ (t'$\in$f(s_{i-1}))
$\Longrightarrow$
$\exists$t s.t. ((t',$\mathtt{COMMIT}_x$,t)$\in$steps(SLM)) $\wedge$ (t$\in$f(s_i))

Proof:

- π_i=$\texttt{COMMIT}_x \implies$ curr_reqd_dlr=x in state s_i
- Either

  (1) $\exists y$, $y \neq x$, s.t. recvbl(y) in state s_{i-1}

  - (recvbl(y) in state s_{i-1}) $\wedge$ (π_i=$\texttt{COMMIT}_x$) $\implies$ recvbl(y) in state s_i
  - ((curr_reqd_dlr=x) $\wedge$ (recvbl(y), $y \neq x$)) in state s_i
    $\implies$ lm_keep(x) in state s_i
  - (recvbl(y) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
    $\implies$ ((let_erase$\neq\emptyset$) $\vee$ (wait_erase$\neq\emptyset$)) in state t'
    by Lemma A.7
  - (((let_erase$\neq\emptyset$) $\vee$ (wait_erase$\neq\emptyset$)) in state t')
    $\wedge$ ((t',$\texttt{COMMIT}_x$,t)$\in$steps($\texttt{SLM}$))
    $\implies$ x=keep in state t
  - Either

    (i) $\exists w$, $w \neq x$, s.t. curr_reqd_dlr=w in state s_{i-1}

    - curr_reqd_dlr=w in state s_{i-1}
      $\implies$ ((recvbl(w)) $\wedge$ (status$_w\neq$RECV)) in state s_{i-1}
      by Invariants 3.6 and 3.8
    - (((recvbl(w)) $\wedge$ (status$_w\neq$RECV)) in state s_{i-1})
      $\wedge$ (π_i=$\texttt{COMMIT}_x$)
      $\implies$ ((recvbl(w)) $\wedge$ (status$_w\neq$RECV)) in state s_i
    - (π_i=$\texttt{COMMIT}_x$) $\wedge$ ($w \neq x$) $\implies$ curr_reqd_dlr$\neq w$ in state s_i
    - ((curr_reqd_dlr$\neq w$)
      $\wedge$ (recvbl(w))
      $\wedge$ (status$_w\neq$RECV)) in state s_i
      $\implies$ lm_let(w) in state s_i
    - curr_reqd_dlr=w in state s_{i-1}
      $\implies$ ((lm_keep(w)) $\vee$ (lm_let(w))) in state s_{i-1}
    - (((lm_keep(w)) $\vee$ (lm_let(w))) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
      $\implies$ ((keep=w) $\vee$ ($w\in$let_erase)) in state t'
      by Lemmas A.2 and A.3
    - (((keep=w) $\vee$ ($w\in$let_erase)) in state t')
      $\wedge$ ((t',$\texttt{COMMIT}_x$,t)$\in$steps($\texttt{SLM}$))
      $\implies$ $w\in$let_erase in state t

    or

    (ii) $\nexists w$, $w \neq x$, s.t. curr_reqd_dlr=w in state s_{i-1}

    - $\nexists w$, $w \neq x$, s.t. curr_reqd_dlr=w in state s_{i-1}
      $\implies$ $\nexists w$, $w \neq x$, s.t. lm_keep(w) in state s_{i-1}
  - $\forall z$, $z \neq x$, (lm_let(z) in state s_{i-1}) $\wedge$ (π_i=$\texttt{COMMIT}_x$)
    $\implies$ lm_let(z) in state s_i
  - $\forall z$, $z \neq x$, (lm_let(z) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
    $\implies$ $z\in$let_erase in state t' by Lemma A.3

- $\forall z,\ z{\neq}x$, ($z{\in}$let_erase in state t') $\wedge$ ((t',$\mathtt{COMMIT}_x$,t)$\in$steps($\mathtt{SLM}$))
  $\Longrightarrow$ $z{\in}$let_erase in state t
- $\forall z,\ z{\neq}x$, (lm_wait(z) in state s_{i-1}) $\wedge$ (π_i=$\mathtt{COMMIT}_x$)
  $\Longrightarrow$ lm_wait(z) in state s_i
- $\forall z,\ z{\neq}x$, (lm_wait(z) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z{\in}$wait_erase in state t' by Lemma A.4
- $\forall z,\ z{\neq}x$, ($z{\in}$wait_erase in state t') $\wedge$ ((t',$\mathtt{COMMIT}_x$,t)$\in$steps($\mathtt{SLM}$))
  $\Longrightarrow$ $z{\in}$wait_erase in state t

or

(2) $\nexists y,\ y{\neq}x$, s.t. recvbl(y) in state s_{i-1}

- ($\nexists y,\ y{\neq}x$, s.t. recvbl(y) in state s_{i-1}) $\wedge$ (π_i=$\mathtt{COMMIT}_x$)
  $\Longrightarrow$ $\nexists y,\ y{\neq}x$, s.t. recvbl(y) in state s_i
- ((curr_reqd_dlr=x) $\wedge$ ($\nexists y,\ y{\neq}x$, s.t. recvbl(y))) in state s_i
  $\Longrightarrow$ lm_let(x) in state s_i
- ($\nexists y,\ y{\neq}x$, s.t. recvbl(y) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ ((let_erase=$\emptyset$) $\wedge$ (wait_erase=$\emptyset$)) in state t'
  by Lemma A.8
- (((let_erase=$\emptyset$) $\wedge$ (wait_erase=$\emptyset$)) in state t')
  $\wedge$ ((t',$\mathtt{COMMIT}_x$,t)$\in$steps($\mathtt{SLM}$))
  $\Longrightarrow$ ((keep=$\perp$) $\wedge$ ($x{\in}$let_erase)) in state t

- $\forall z,\ z{\neq}x$, ($\neg$recvbl(z) in state s_{i-1}) $\wedge$ (π_i=$\mathtt{COMMIT}_x$) $\Longrightarrow$ $\neg$recvbl(z) in state s_i
- $\forall z,\ z{\neq}x$, ($\neg$recvbl(z) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ ((keep$\neq z$) $\wedge$ ($z{\notin}$let_erase) $\wedge$ ($z{\notin}$wait_erase)) in state t'
  by Lemma A.5
- $\forall z,\ z{\neq}x$, (((keep$\neq z$) $\wedge$ ($z{\notin}$let_erase) $\wedge$ ($z{\notin}$wait_erase)) in state t')
  $\wedge$ ((t',$\mathtt{COMMIT}_x$,t)$\in$steps($\mathtt{SLM}$))
  $\Longrightarrow$ ((keep$\neq z$) $\wedge$ ($z{\notin}$let_erase) $\wedge$ ($z{\notin}$wait_erase)) in state t
- Therefore, from the above deductions it follows that
  t$\in$f(s_i) by Definition 3.1

and thus the theorem has been proven. □

Theorem A.10

$$
\begin{array}{ll}
 & (s_{i-1} \text{ is a reachable state of } \mathtt{LM}) \\
\wedge & (\pi_i{=}\mathtt{ERASABLE}_x) \\
\wedge & ((s_{i-1},\mathtt{ERASABLE}_x,s_i)\in \mathrm{steps}(\mathtt{LM})) \\
\wedge & (t'\in f(s_{i-1})) \\
\Longrightarrow & \\
 & \exists t \text{ s.t. } ((t',\mathtt{ERASABLE}_x,t)\in \mathrm{steps}(\mathtt{SLM})) \wedge (t\in f(s_i))
\end{array}
$$

Proof:

- π_i=$\mathtt{ERASABLE}_x$ $\Longrightarrow$ $x{\in}$pending_erasable in state s_{i-1}
- $x{\in}$pending_erasable in state s_{i-1}
  $\Longrightarrow$ ((status$_x$=RECV) $\wedge$ (pending_ack$_x$=F)) in state s_{i-1}
  by Invariants 3.14 and 3.12

- $(((\text{status}_x{=}\text{RECV}) \land (\text{pending\_ack}_x{=}\text{F})) \text{ in state } s_{i-1}) \land (\pi_i{=}\texttt{ERASABLE}_x)$
  $\implies \text{lm\_wait}(x) \text{ in state } s_i$
- $x{\in}\text{pending\_erasable in state } s_{i-1}$
  $\implies ((\text{status}_x{=}\text{RECV}) \land (\text{curr\_reqd\_dlr}{\neq}x) \land (\text{timestamp}_x{\in}\mathcal{N}))$
  in state s_{i-1}
  by Invariants 3.14, 3.6 and 3.7
- $($ $(\text{status}_x{=}\text{RECV})$
  $\land$ $(\text{curr\_reqd\_dlr}{\neq}x)$
  $\land$ $(\text{timestamp}_x{\in}\mathcal{N})$
  $\land$ $(x{\in}\text{pending\_erasable}))$ in state s_{i-1}
  $\implies \text{lm\_let}(x) \text{ in state } s_{i-1}$ by definition of $\text{lm\_let}(x)$
- $(\text{lm\_let}(x) \text{ in state } s_{i-1}) \land (\text{t'}{\in}\text{f}(s_{i-1})) \implies x{\in}\text{let\_erase in state t'}$
  by Lemma A.3
- $(x{\in}\text{let\_erase in state t'}) \land ((\text{t'},\texttt{ERASABLE}_x,\text{t}){\in}\text{steps}(\texttt{SLM}))$
  $\implies x{\in}\text{wait\_erase in state t}$
- $\forall z,\ z{\neq}x,\ (\text{lm\_keep}(z) \text{ in state } s_{i-1}) \land (\pi_i{=}\texttt{ERASABLE}_x)$
  $\implies \text{lm\_keep}(z) \text{ in state } s_i$
- $\forall z,\ z{\neq}x,\ (\text{lm\_keep}(z) \text{ in state } s_{i-1}) \land (\text{t'}{\in}\text{f}(s_{i-1}))$
  $\implies \text{keep}{=}z \text{ in state t'}$ by Lemma A.3
- $\forall z,\ z{\neq}x,\ (\text{keep}{=}z \text{ in state t'}) \land ((\text{t'},\texttt{ERASABLE}_x,\text{t}){\in}\text{steps}(\texttt{SLM}))$
  $\implies \text{keep}{=}z \text{ in state t}$
- $\forall z,\ z{\neq}x,\ (\text{lm\_let}(z) \text{ in state } s_{i-1}) \land (\pi_i{=}\texttt{ERASABLE}_x) \implies \text{lm\_let}(z) \text{ in state } s_i$
- $\forall z,\ z{\neq}x,\ (\text{lm\_let}(z) \text{ in state } s_{i-1}) \land (\text{t'}{\in}\text{f}(s_{i-1}))$
  $\implies z{\in}\text{let\_erase in state t'}$ by Lemma A.3
- $\forall z,\ z{\neq}x,\ (z{\in}\text{let\_erase in state t'}) \land ((\text{t'},\texttt{ERASABLE}_x,\text{t}){\in}\text{steps}(\texttt{SLM}))$
  $\implies z{\in}\text{let\_erase in state t}$
- $\forall z,\ z{\neq}x,\ (\text{lm\_wait}(z) \text{ in state } s_{i-1}) \land (\pi_i{=}\texttt{ERASABLE}_x)$
  $\implies \text{lm\_wait}(z) \text{ in state } s_i$
- $\forall z,\ z{\neq}x,\ (\text{lm\_wait}(z) \text{ in state } s_{i-1}) \land (\text{t'}{\in}\text{f}(s_{i-1}))$
  $\implies z{\in}\text{wait\_erase in state t'}$ by Lemma A.4
- $\forall z,\ z{\neq}x,\ (z{\in}\text{wait\_erase in state t'}) \land ((\text{t'},\texttt{ERASABLE}_x,\text{t}){\in}\text{steps}(\texttt{SLM}))$
  $\implies z{\in}\text{wait\_erase in state t}$
- $\forall z,\ z{\neq}x,\ (\neg\text{recvbl}(z) \text{ in state } s_{i-1}) \land (\pi_i{=}\texttt{ERASABLE}_x)$
  $\implies \neg\text{recvbl}(z) \text{ in state } s_i$
- $\forall z,\ z{\neq}x,\ (\neg\text{recvbl}(z) \text{ in state } s_{i-1}) \land (\text{t'}{\in}\text{f}(s_{i-1}))$
  $\implies ((\text{keep}{\neq}z) \land (z{\notin}\text{let\_erase}) \land (z{\notin}\text{wait\_erase})) \text{ in state t'}$
  by Lemma A.5
- $\forall z,\ z{\neq}x,$ $(((\text{keep}{\neq}z) \land (z{\notin}\text{let\_erase}) \land (z{\notin}\text{wait\_erase})) \text{ in state t'})$
  $\land$ $((\text{t'},\texttt{ERASABLE}_x,\text{t}){\in}\text{steps}(\texttt{SLM}))$
  $\implies ((\text{keep}{\neq}z) \land (z{\notin}\text{let\_erase}) \land (z{\notin}\text{wait\_erase})) \text{ in state t}$
- Therefore, from the above deductions it follows that
  $\text{t}{\in}\text{f}(s_i)$ by Definition 3.1
  and thus the theorem has been proven. □

Theorem A.11

$$\begin{array}{ll} & (\mathrm{s}_{i-1} \text{ is a reachable state of } \texttt{LM}) \\ \wedge & (\pi_i{=}\texttt{ERASE}_x) \\ \wedge & ((\mathrm{s}_{i-1},\texttt{ERASE}_x,\mathrm{s}_i){\in}\mathrm{steps}(\texttt{LM})) \\ \wedge & (\mathrm{t'}{\in}\mathrm{f}(\mathrm{s}_{i-1})) \\ \Longrightarrow & \\ & \exists \mathrm{t} \text{ s.t. } ((\mathrm{t'},\texttt{ERASE}_x,\mathrm{t}){\in}\mathrm{steps}(\texttt{SLM})) \wedge (\mathrm{t}{\in}\mathrm{f}(\mathrm{s}_i)) \end{array}$$

Proof:

- π_i=$\texttt{ERASE}_x$ $\Longrightarrow$ $x\in$can_erase in state s_{i-1}
- $x\in$can_erase in state s_{i-1}
  $\Longrightarrow$ lm_wait(x) in state s_{i-1} by Invariants 3.14, 3.13 and 3.12
- lm_wait(x) in state s_{i-1}
  $\Longrightarrow$ $\nexists v$ s.t. $<x,v>\in$pending_ts_assign in state s_{i-1} by Invariant 3.9
- ($\nexists v$ s.t. $<x,v>\in$pending_ts_assign in state s_{i-1}) $\wedge$ (π_i=$\texttt{ERASE}_x$)
  $\Longrightarrow$ $\neg$recvbl(x) in state s_i
- (lm_wait(x) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $x\in$wait_erase in state t' by Lemma A.4
- $x\in$wait_erase in state t'
  $\Longrightarrow$ ((keep$\neq x$) $\wedge$ ($x\notin$let_erase)) in state t' by Invariant 3.4
- (((keep$\neq x$) $\wedge$ ($x\notin$let_erase)) in state t') $\wedge$ ((t',$\texttt{ERASE}_x$,t)$\in$steps($\texttt{SLM}$))
  $\Longrightarrow$ ((keep$\neq x$) $\wedge$ ($x\notin$let_erase) $\wedge$ ($x\notin$wait_erase)) in state t
- Either
  (1) $\exists w$, $w\in\mathcal{N}$, s.t. curr_reqd_dlr=w in state s_{i-1}
    - (curr_reqd_dlr=w in state s_{i-1}) $\wedge$ (π_i=$\texttt{ERASE}_x$)
      $\Longrightarrow$ curr_reqd_dlr=w in state s_i
    - ((curr_reqd_dlr=w) $\wedge$ (status_x=RECV)) in state s_{i-1}
      $\Longrightarrow$ $w\neq x$ by definition of lm_wait(x) and Invariant 3.6
    - lm_wait(x) in state s_{i-1}
      $\Longrightarrow$ recvbl(x) in state s_{i-1} by Invariant 3.7
    - ((curr_reqd_dlr=w) $\wedge$ (recvbl(x)) $\wedge$ ($w\neq x$)) in state s_{i-1}
      $\Longrightarrow$ lm_keep(w) in state s_{i-1}
    - (lm_keep(w) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
      $\Longrightarrow$ keep=w in state t' by Lemma A.2
    - Either
      (i) $\exists y$, $y\notin\{x,w\}$, s.t. recvbl(y) in state s_{i-1}
        - (recvbl(y) in state s_{i-1}) $\wedge$ (π_i=$\texttt{ERASE}_x$) $\wedge$ ($y\neq x$)
          $\Longrightarrow$ recvbl(y) in state s_i
        - ((curr_reqd_dlr=w) $\wedge$ ($\exists y$, $y\neq w$, s.t. recvbl(y))) in state s_i
          $\Longrightarrow$ lm_keep(w) in state s_i
          by definition of lm_keep(w)

- (recvbl(y) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ ((keep=y) $\vee$ ($y\in$let_erase) $\vee$ ($y\in$wait_erase))
  in state t'
  by Lemma A.6
- (((keep=y)
  $\vee$ ($y\in$let_erase)
  $\vee$ ($y\in$wait_erase)) in state t')
  $\wedge$ (keep=w in state t')
  $\wedge$ ($y\neq w$)
  $\Longrightarrow$ (($y\in$let_erase) $\vee$ ($y\in$wait_erase)) in state t'
- (((keep=w)
  $\wedge$ (($y\in$let_erase) $\vee$ ($y\in$wait_erase))) in state t')
  $\wedge$ ((t',$\texttt{ERASE}_x$,t)$\in$steps($\texttt{SLM}$))
  $\Longrightarrow$ keep=w in state t

or

(ii) $\nexists y$, $y\notin\{x,w\}$, s.t. recvbl(y) in state s_{i-1}

- ($\nexists y$, $y\notin\{x,w\}$, s.t. recvbl(y) in state s_{i-1}) $\wedge$ (π_i=$\texttt{ERASE}_x$)
  $\Longrightarrow$ $\nexists y$, $y\neq w$, s.t. recvbl(y) in state s_i
- ((curr_reqd_dlr=w) $\wedge$ ($\nexists y$, $y\neq w$, s.t. recvbl(y))) in state s_i
  $\Longrightarrow$ lm_let(w) in state s_i
- $x\in$wait_erase in state t'
  $\Longrightarrow$ $x\notin$let_erase in state t' by Invariant 3.4
- keep=w in state t'
  $\Longrightarrow$ (($w\notin$let_erase) $\wedge$ ($w\notin$wait_erase)) in state t'
  by Invariants 3.4 and 3.3
- ($\nexists y$, $y\notin\{x,w\}$, s.t. recvbl(y) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $\forall y$, $y\notin\{x,w\}$, ((keep$\neq y$)
  $\wedge$ ($y\notin$let_erase)
  $\wedge$ ($y\notin$wait_erase)) in state t'
  by Lemma A.5
- (($x\notin$let_erase) $\wedge$ ($x\in$wait_erase) in state t')
  $\wedge$ ((($w\notin$let_erase) $\vee$ ($w\notin$wait_erase)) in state t')
  $\wedge$ ($\forall y$, $y\notin\{x,w\}$, ((keep$\neq y$)
  $\wedge$ ($y\notin$let_erase)
  $\wedge$ ($y\notin$wait_erase)) in state t')
  $\Longrightarrow$ ((let_erase=$\emptyset$) $\wedge$ (wait_erase=$\{x\}$)) in state t'
- (((keep=w)
  $\wedge$ (let_erase=$\emptyset$)
  $\wedge$ (wait_erase=$\{x\}$)) in state t')
  $\wedge$ ((t',$\texttt{ERASE}_x$,t)$\in$steps($\texttt{SLM}$))
  $\Longrightarrow$ ((keep=$\perp$) $\wedge$ ($w\in$let_erase)) in state t

or

(2) $\nexists w$, $w\in\mathcal{N}$, s.t. curr_reqd_dlr=w in state s_{i-1}

- $\nexists w$, $w \in \mathcal{N}$, s.t. curr_reqd_dlr$=w$ in state s_{i-1}
  $\Longrightarrow$ $\nexists w$, $w \in \mathcal{N}$, s.t. lm_keep(z) in state s_{i-1}

- $\forall z$, $z \neq x$, (lm_let(z) in state s_{i-1}) $\wedge$ (π_i=$\texttt{ERASE}_x$) $\Longrightarrow$ lm_let(z) in state s_i
- $\forall z$, $z \neq x$, (lm_let(z) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$let_erase in state t' by Lemma A.3
- $\forall z$, $z \neq x$, ($z\in$let_erase in state t') $\wedge$ ((t',$\texttt{ERASE}_x$,t)$\in$steps($\texttt{SLM}$))
  $\Longrightarrow$ $z\in$let_erase in state t
- $\forall z$, $z \neq x$, (lm_wait(z) in state s_{i-1}) $\wedge$ (π_i=$\texttt{ERASE}_x$) $\Longrightarrow$ lm_wait(z) in state s_i
- $\forall z$, $z \neq x$, (lm_wait(z) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$wait_erase in state t' by Lemma A.4
- $\forall z$, $z \neq x$, ($z\in$wait_erase in state t') $\wedge$ ((t',$\texttt{ERASE}_x$,t)$\in$steps($\texttt{SLM}$))
  $\Longrightarrow$ $z\in$wait_erase in state t
- $\forall z$, $z \neq x$, ($\neg$recvbl(z) in state s_{i-1}) $\wedge$ (π_i=$\texttt{ERASE}_x$) $\Longrightarrow$ $\neg$recvbl(z) in state s_i
- $\forall z$, $z \neq x$, ($\neg$recvbl(z) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ ((keep$\neq z$) $\wedge$ ($z\notin$let_erase) $\wedge$ ($z\notin$wait_erase)) in state t'
  by Lemma A.5
- $\forall z$, $z \neq x$, (((keep$\neq z$) $\wedge$ ($z\notin$let_erase) $\wedge$ ($z\notin$wait_erase)) in state t')
  $\wedge$ ((t',$\texttt{ERASE}_x$,t)$\in$steps($\texttt{SLM}$))
  $\Longrightarrow$ ((keep$\neq z$) $\wedge$ ($z\notin$let_erase) $\wedge$ ($z\notin$wait_erase)) in state t
- Therefore, from the above deductions it follows that
  t$\in$f(s_i) by Definition 3.1
  and thus the theorem has been proven. □

Lemma A.12 (s_{i-1} is a reachable state of $\texttt{LM}$)
$\wedge$ (lm_keep(z) in state s_{i-1})
$\wedge$ (π_i=$\texttt{ACK\_ASSIGN}_x$)
$\Longrightarrow$
lm_keep(z) in state s_i

Proof:

- lm_keep(z) in state s_{i-1}
  $\Longrightarrow$ ((curr_reqd_dlr$=z$) $\wedge$ ($\exists y$, $y \neq z$, s.t. recvbl(y))) in state s_{i-1}
- ((curr_reqd_dlr$=z$) $\wedge$ ($\exists y$, $y \neq z$, s.t. recvbl(y))) in state s_{i-1}
  $\Longrightarrow$ recv_tss$\neq\emptyset$ in state s_{i-1} by Invariant 3.17
- (((curr_reqd_dlr$=z$) $\wedge$ (recv_tss$\neq\emptyset$)) in state s_{i-1}) $\wedge$ (π_i=$\texttt{ACK\_ASSIGN}_x$)
  $\Longrightarrow$ curr_reqd_dlr$=z$ in state s_i
- (recvbl(y) in state s_{i-1}) $\wedge$ (π_i=$\texttt{ACK\_ASSIGN}_x$) $\Longrightarrow$ recvbl(y) in state s_i
- ((curr_reqd_dlr$=z$) $\wedge$ (recvbl(y), $y \neq z$)) in state s_i $\Longrightarrow$ lm_keep(z) in state s_i
  and thus the lemma has been proven. □

Lemma A.13

$$
\begin{aligned}
& (s_{i-1} \text{ is a reachable state of } \mathtt{LM}) \\
\wedge\ & (\text{lm\_let}(z) \text{ in state } s_{i-1}) \\
\wedge\ & (\pi_i = \mathtt{ACK\_ASSIGN}_x) \\
\Longrightarrow\ & \\
& \text{lm\_let}(z) \text{ in state } s_i
\end{aligned}
$$

Proof:

- lm_let(z) in state s_{i-1}

  $\Longrightarrow$ Either

  (1) ((curr_reqd_dlr=z) $\wedge$ ($\nexists y$, $y \neq z$, s.t. recvbl(y))) in state s_{i-1}

  - ($\nexists y$, $y \neq z$, s.t. recvbl(y) in state s_{i-1}) $\wedge$ (π_i=$\mathtt{ACK\_ASSIGN}_x$)

    $\Longrightarrow$ $\nexists y$, $y \neq z$, s.t. recvbl(y) in state s_i
  - curr_reqd_dlr=z in state s_{i-1}

    $\Longrightarrow$ ((recvbl(z)) $\wedge$ (status$_z$$\neq$RECV)) in state s_{i-1}

    by Invariants 3.6 and 3.8
  - (((recvbl(z)) $\wedge$ (status$_z$$\neq$RECV)) in state s_{i-1})

    $\wedge$ (π_i=$\mathtt{ACK\_ASSIGN}_x$)

    $\Longrightarrow$ ((recvbl(z)) $\wedge$ (status$_z$$\neq$RECV)) in state s_i
  - (($\nexists y$, $y \neq z$, s.t. recvbl(y))

    $\wedge$ (recvbl(z))

    $\wedge$ (status$_z$$\neq$RECV)) in state s_i

    $\Longrightarrow$ lm_let(z) in state s_i

  or

  (2) ((curr_reqd_dlr$\neq$$z$) $\wedge$ (recvbl(z)) $\wedge$ ($\neg$lm_wait(z))) in state s_{i-1}

  - (((curr_reqd_dlr$\neq$$z$)

    $\wedge$ (recvbl(z))

    $\wedge$ ($\neg$lm_wait(z))) in state s_{i-1})

    $\wedge$ (π_i=$\mathtt{ACK\_ASSIGN}_x$)

    $\Longrightarrow$ ((curr_reqd_dlr$\neq$$z$) $\wedge$ (recvbl(z)) $\wedge$ ($\neg$lm_wait(z)))

    in state s_i
  - ((curr_reqd_dlr$\neq$$z$) $\wedge$ (recvbl(z)) $\wedge$ ($\neg$lm_wait(z))) in state s_i

    $\Longrightarrow$ lm_let(z) in state s_i by definition of lm_let(z)
- For both possible cases, the desired result is obtained and thus the lemma has been proven. □

Theorem A.14

$$
\begin{aligned}
& (s_{i-1} \text{ is a reachable state of } \mathtt{LM}) \\
\wedge\ & (\pi_i = \mathtt{ACK\_ASSIGN}_x) \\
\wedge\ & ((s_{i-1}, \mathtt{ACK\_ASSIGN}_x, s_i) \in \text{steps}(\mathtt{LM})) \\
\wedge\ & (t' \in f(s_{i-1})) \\
\Longrightarrow\ & \\
& t' \in f(s_i)
\end{aligned}
$$

Proof:

- $\forall z,\ z \in \mathcal{N}$, (lm_keep($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ keep=z in state t' — by Lemma A.2
- $\forall z,\ z \in \mathcal{N}$, (lm_keep($z$) in state s_{i-1}) $\wedge$ (π_i=`ACK_ASSIGN`$_x$)
  $\Longrightarrow$ lm_keep(z) in state s_i — by Lemma A.12
- $\forall z,\ z \in \mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$let_erase in state t' — by Lemma A.3
- $\forall z,\ z \in \mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (π_i=`ACK_ASSIGN`$_x$)
  $\Longrightarrow$ lm_let(z) in state s_i — by Lemma A.13
- $\forall z,\ z \in \mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$wait_erase in state t' — by Lemma A.4
- $\forall z,\ z \in \mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (π_i=`ACK_ASSIGN`$_x$)
  $\Longrightarrow$ lm_wait(z) in state s_i
- $\forall z,\ z \in \mathcal{N}$, ($\neg$recvbl($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ ((keep$\neq z$) $\wedge$ ($z\notin$let_erase) $\wedge$ ($z\in$wait_erase)) in state t'
  by Lemma A.5
- $\forall z,\ z \in \mathcal{N}$, ($\neg$recvbl($z$) in state s_{i-1}) $\wedge$ (π_i=`ACK_ASSIGN`$_x$)
  $\Longrightarrow$ $\neg$recvbl(z) in state s_i
- Therefore, from the above deductions it follows that
  t'$\in$f(s_i) — by Definition 3.1

  and thus the theorem has been proven. □

Theorem A.15

$$
\begin{array}{ll}
 & (s_{i-1} \text{ is a reachable state of } \mathtt{LM}) \\
\wedge & (\pi_i = \mathtt{ACK\_CS\_RECV}_x) \\
\wedge & ((s_{i-1}, \mathtt{ACK\_CS\_RECV}_x, s_i) \in \text{steps}(\mathtt{LM})) \\
\wedge & (t' \in f(s_{i-1})) \\
\Longrightarrow & \\
 & t' \in f(s_i)
\end{array}
$$

Proof:

- $\forall z,\ z \in \mathcal{N}$, (lm_keep($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ keep=z in state t' — by Lemma A.2
- $\forall z,\ z \in \mathcal{N}$,
  (lm_keep(z) in state s_{i-1}) $\wedge$ (π_i=`ACK_CS_RECV`$_x$) $\Longrightarrow$ lm_keep(z) in state s_i
- $\forall z,\ z \in \mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$let_erase in state t' — by Lemma A.3
- $\forall z,\ z \in \mathcal{N}$,
  (lm_let(z) in state s_{i-1}) $\wedge$ (π_i=`ACK_CS_RECV`$_x$) $\Longrightarrow$ lm_let(z) in state s_i
- $\forall z,\ z \in \mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$wait_erase in state t' — by Lemma A.4
- $\forall z,\ z \in \mathcal{N}$,
  (lm_wait(z) in state s_{i-1}) $\wedge$ (π_i=`ACK_CS_RECV`$_x$) $\Longrightarrow$ lm_wait(z) in state s_i

- $\forall z,\ z\in\mathcal{N}$, ($\neg$recvbl($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ ((keep$\neq z$) $\wedge$ ($z\notin$let_erase) $\wedge$ ($z\in$wait_erase)) in state t'
  by Lemma A.5
- $\forall z,\ z\in\mathcal{N}$,
  ($\neg$recvbl(z) in state s_{i-1}) $\wedge$ (π_i=`ACK_CS_RECV`$_x$) $\Longrightarrow$ $\neg$recvbl(z) in state s_i
- Therefore, from the above deductions it follows that
  t'$\in$f(s_i) by Definition 3.1
  and thus the theorem has been proven. □

Lemma A.16 (s_{i-1} is a reachable state of `LM`)
$\wedge$ (lm_keep(z) in state s_{i-1})
$\wedge$ (π_i=<`DLR_GONE`,u>)
$\Longrightarrow$
lm_keep(z) in state s_i

Proof:

- lm_keep(z) in state s_{i-1}
  $\Longrightarrow$ ((curr_reqd_dlr=z) $\wedge$ ($\exists y,\ y\neq z$, s.t. recvbl(y))) in state s_{i-1}
  by definition of lm_keep(z)
- ((curr_reqd_dlr$\neq y$) $\wedge$ (recvbl(y))) in state s_{i-1}
  $\Longrightarrow$ $\exists v,\ v\in\mathcal{N}$, s.t.
  (((<y,v>$\in$pending_ts_assign) $\vee$ (timestamp$_y$=v))
  $\wedge$ ($v\in$recv_tss)) in state s_{i-1}
  by Invariant 3.17
- π_i=<`DLR_GONE`,u>
  $\Longrightarrow$ $\exists x$ s.t. ((status$_x$=NONR) $\wedge$ (timestamp$_x$=u)) in state s_{i-1}
- status$_x$=NONR in state s_{i-1}
  $\Longrightarrow$ $\nexists w$ s.t. <x,w>$\in$pending_ts_assign in state s_{i-1} by Invariant 3.9
- (($\nexists w$ s.t. <x,w>$\in$pending_ts_assign) $\wedge$ (status$_x$=NONR)) in state s_{i-1}
  $\Longrightarrow$ $\neg$recvbl(x) in state s_{i-1}
- ((recvbl(y)) $\wedge$ ($\neg$recvbl(x))) in state s_{i-1} $\Longrightarrow$ $x\neq y$
- (((<y,v>$\in$pending_ts_assign) $\vee$ (timestamp$_y$=v, $v\in\mathcal{N}$))
  $\wedge$ (timestamp$_x$=u)
  $\wedge$ ($x\neq y$)) in state s_{i-1}
  $\Longrightarrow$ $u\neq v$ by Invariant 3.16
- (((curr_reqd_dlr=z) $\wedge$ ($v\in$recv_tss)) in state s_{i-1})
  $\wedge$ (π_i=<`DLR_GONE`,u>)
  $\wedge$ ($u\neq v$)
  $\Longrightarrow$ curr_reqd_dlr=z in state s_i
- ($\exists y,\ y\neq z$, s.t. recvbl(y) in state s_{i-1}) $\wedge$ (π_i=<`DLR_GONE`,u>)
  $\Longrightarrow$ $\exists y,\ y\neq z$, s.t. recvbl(y) in state s_i
- ((curr_reqd_dlr=z) $\wedge$ ($\exists y,\ y\neq z$, s.t. recvbl(y))) in state s_i
  $\Longrightarrow$ lm_keep(z) in state s_i
  and thus the lemma has been proven. □

Lemma A.17
$$
\begin{aligned}
& (s_{i-1} \text{ is a reachable state of } \texttt{LM}) \\
\wedge\ & (\text{lm\_let}(z) \text{ in state } s_{i-1}) \\
\wedge\ & (\pi_i = <\texttt{DLR\_GONE}, u>) \\
\Longrightarrow\ & \\
& \text{lm\_let}(z) \text{ in state } s_i
\end{aligned}
$$

Proof:

- lm_let(z) in state s_{i-1}

  $\Longrightarrow$ Either

  (1) $((\text{curr\_reqd\_dlr}=z) \wedge (\nexists y,\ y \neq z, \text{ s.t. recvbl}(y)))$ in state s_{i-1}

  - curr_reqd_dlr$=z$ in state s_{i-1}

    $\Longrightarrow$ $((\text{recvbl}(z)) \wedge (\text{status}_z \neq \text{RECV}))$ in state s_{i-1}

    by Invariants 3.6 and 3.8

  - $$
\begin{aligned}
& ((\ (\nexists y,\ y \neq z, \text{ s.t. recvbl}(y)) \\
& \quad \wedge\ (\text{recvbl}(z)) \\
& \quad \wedge\ (\text{status}_z \neq \text{RECV})) \quad \text{in state } s_{i-1}) \\
\wedge\ & (\pi_i = <\texttt{DLR_GONE}, u>) \\
& \Longrightarrow (\ (\nexists y,\ y \neq z, \text{ s.t. recvbl}(y)) \\
& \qquad \wedge\ (\text{recvbl}(z)) \\
& \qquad \wedge\ (\text{status}_z \neq \text{RECV})) \quad \text{in state } s_i
\end{aligned}
$$

  - $$
\begin{aligned}
& (\ (\nexists y,\ y \neq z, \text{ s.t. recvbl}(y)) \\
& \wedge\ (\text{recvbl}(z)) \\
& \wedge\ (\text{status}_z \neq \text{RECV})\ ) \text{ in state } s_i
\end{aligned}
$$

    $\Longrightarrow$ lm_let(z) in state s_i by definition of lm_let(z)

  or

  (2) $((\text{curr\_reqd\_dlr} \neq z) \wedge (\text{recvbl}(z)) \wedge (\neg\text{lm\_wait}(z)))$ in state s_{i-1}

  - $$
\begin{aligned}
& ((\ (\text{curr_reqd_dlr} \neq z) \\
& \quad \wedge\ (\text{recvbl}(z)) \\
& \quad \wedge\ (\neg\text{lm_wait}(z))\ ) \text{ in state } s_{i-1}) \\
\wedge\ & (\pi_i = <\texttt{DLR_GONE}, u>)
\end{aligned}
$$

    $\Longrightarrow$ $((\text{curr\_reqd\_dlr} \neq z) \wedge (\text{recvbl}(z)) \wedge (\neg\text{lm\_wait}(z)))$ in state s_i

  - $((\text{curr\_reqd\_dlr} \neq z) \wedge (\text{recvbl}(z)) \wedge (\neg\text{lm\_wait}(z)))$ in state s_i

    $\Longrightarrow$ lm_let(z) in state s_i by definition of lm_let(z)

- For both possible cases, the desired result is obtained and thus the lemma has been proven. □

Theorem A.18
$$
\begin{aligned}
& (s_{i-1} \text{ is a reachable state of } \texttt{LM}) \\
\wedge\ & (\pi_i = <\texttt{DLR_GONE}, u>) \\
\wedge\ & ((s_{i-1}, <\texttt{DLR_GONE}, u>, s_i) \in \text{steps}(\texttt{LM})) \\
\wedge\ & (t' \in f(s_{i-1})) \\
\Longrightarrow\ & \\
& t' \in f(s_i)
\end{aligned}
$$

Proof:

- $\forall z,\ z\in\mathcal{N}$, (lm_keep($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ keep=z in state t' by Lemma A.2
- $\forall z,\ z\in\mathcal{N}$, (lm_keep($z$) in state s_{i-1}) $\wedge$ (π_i=<`DLR_GONE`,u>)
  $\Longrightarrow$ lm_keep(z) in state s_i by Lemma A.16
- $\forall z,\ z\in\mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$let_erase in state t' by Lemma A.3
- $\forall z,\ z\in\mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (π_i=<`DLR_GONE`,u>)
  $\Longrightarrow$ lm_let(z) in state s_i by Lemma A.17
- $\forall z,\ z\in\mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$wait_erase in state t' by Lemma A.4
- $\forall z,\ z\in\mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (π_i=<`DLR_GONE`,u>)
  $\Longrightarrow$ lm_wait(z) in state s_i
- $\forall z,\ z\in\mathcal{N}$, ($\neg$recvbl($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ ((keep$\neq z$) $\wedge$ ($z\notin$let_erase) $\wedge$ ($z\in$wait_erase)) in state t'
  by Lemma A.5
- $\forall z,\ z\in\mathcal{N}$, ($\neg$recvbl($z$) in state s_{i-1}) $\wedge$ (π_i=<`DLR_GONE`,u>)
  $\Longrightarrow$ $\neg$recvbl(z) in state s_i
- Therefore, from the above deductions it follows that
  t'$\in$f(s_i) by Definition 3.1
  and thus the theorem has been proven. $\square$

Lemma A.19 (s_{i-1} is a reachable state of `LM`)
$\wedge$ (lm_keep(z) in state s_{i-1})
$\wedge$ (π_i=<`ASSIGN`$_x$,v>)
$\Longrightarrow$
lm_keep(z) in state s_i

Proof:

- lm_keep(z) in state s_{i-1}
  $\Longrightarrow$ ((curr_reqd_dlr=z) $\wedge$ ($\exists y,\ y\neq z$, s.t. recvbl(y))) in state s_{i-1}
- (curr_reqd_dlr=z in state s_{i-1}) $\wedge$ (π_i=<`ASSIGN`$_x$,v>)
  $\Longrightarrow$ curr_reqd_dlr=z in state s_i
- π_i=<`ASSIGN`$_x$,v> $\Longrightarrow$ <x,v>$\in$pending_ts_assign in state s_{i-1}
- <x,v>$\in$pending_ts_assign in state s_{i-1}
  $\Longrightarrow$ ($v\in\mathcal{N}$) $\wedge$ (status$_x$=UNFL in state s_{i-1}) by Invariant 3.9
- (status$_x$=UNFL in state s_{i-1}) $\wedge$ (π_i=<`ASSIGN`$_x$,v>) $\wedge$ ($v\in\mathcal{N}$)
  $\Longrightarrow$ ((timestamp$_x\in\mathcal{N}$) $\wedge$ (status$_x$=UNFL)) in state s_i
- ((timestamp$_x\in\mathcal{N}$) $\wedge$ (status$_x$=UNFL)) in state s_i
  $\Longrightarrow$ recvbl(x) in state s_i by definition of recvbl(x)
- (recvbl(y) in state s_{i-1}) $\wedge$ (π_i=<`ASSIGN`$_x$,v>) $\Longrightarrow$ recvbl(y) in state s_i

- ((curr_reqd_dlr=z) $\wedge$ ($\exists y$, $y{\neq}z$, s.t. recvbl(y))) in state s_i
  $\Longrightarrow$ lm_keep(z) in state s_i
  and thus the lemma has been proven. $\square$

Lemma A.20 (s$_{i-1}$ is a reachable state of `LM`)
$\wedge$ (lm_let(z) in state s_{i-1})
$\wedge$ (π_i=<$\texttt{ASSIGN}_x$,v>)
$\Longrightarrow$
lm_let(z) in state s_i

Proof:

- π_i=<$\texttt{ASSIGN}_x$,v> $\Longrightarrow$ <x,v>$\in$pending_ts_assign in state s_{i-1}
- <x,v>$\in$pending_ts_assign in state s_{i-1}
  $\Longrightarrow$ ($v\in\mathcal{N}$) $\wedge$ (status$_x$=UNFL in state s_{i-1}) by Invariant 3.9
- (status$_x$=UNFL in state s_{i-1}) $\wedge$ (π_i=<$\texttt{ASSIGN}_x$,v>) $\wedge$ ($v\in\mathcal{N}$)
  $\Longrightarrow$ ((timestamp$_x\in\mathcal{N}$) $\wedge$ (status$_x$=UNFL)) in state s_i
- lm_let(z) in state s_{i-1}
  $\Longrightarrow$ Either
  (1) ((curr_reqd_dlr=z) $\wedge$ ($\not\exists y$, $y{\neq}z$, s.t. recvbl(y))) in state s_{i-1}
    - ((<x,v>$\in$pending_ts_assign)
      $\wedge$ ($\not\exists y$, $y{\neq}z$, s.t. recvbl(y))) in state s_{i-1}
      $\Longrightarrow$ x=z
    - (curr_reqd_dlr=z in state s_{i-1}) $\wedge$ (π_i=<$\texttt{ASSIGN}_z$,v>)
      $\Longrightarrow$ curr_reqd_dlr=z in state s_i
    - ($\not\exists y$, $y{\neq}z$, s.t. recvbl(y) in state s_{i-1}) $\wedge$ (π_i=<$\texttt{ASSIGN}_z$,v>)
      $\Longrightarrow$ $\not\exists y$, $y{\neq}z$, s.t. recvbl(y) in state s_i
    - ((curr_reqd_dlr=z) $\wedge$ ($\not\exists y$, $y{\neq}z$, s.t. recvbl(y))) in state s_i
      $\Longrightarrow$ lm_let(z) in state s_i

  or

  (2) ((curr_reqd_dlr$\neq z$) $\wedge$ (recvbl(z)) $\wedge$ ($\neg$lm_wait(z))) in state s_{i-1}
    - (((curr_reqd_dlr$\neq z$)
      $\wedge$ (recvbl(z))
      $\wedge$ ($\neg$lm_wait(z))) in state s_{i-1})
      $\wedge$ (π_i=<$\texttt{ASSIGN}_x$,v>)
      $\wedge$ ($v\in\mathcal{N}$)
      $\Longrightarrow$ ((curr_reqd_dlr$\neq z$) $\wedge$ (recvbl(z)) $\wedge$ ($\neg$lm_wait(z)))
      in state s_i
    - ((curr_reqd_dlr$\neq z$) $\wedge$ (recvbl(z)) $\wedge$ ($\neg$lm_wait(z))) in state s_i
      $\Longrightarrow$ lm_let(z) in state s_i
- For both possible cases, the desired result is obtained and thus the lemma has been proven. $\square$

Lemma A.21

$$\begin{array}{rl} & (s_{i-1} \text{ is a reachable state of } \mathtt{LM}) \\ \wedge & (\text{lm\_wait}(z) \text{ in state } s_{i-1}) \\ \wedge & (\pi_i{=}{<}\mathtt{ASSIGN}_x,v{>}) \\ \Longrightarrow & \\ & \text{lm\_wait}(z) \text{ in state } s_i \end{array}$$

Proof:

- $\pi_i{=}{<}\mathtt{ASSIGN}_x,v{>} \Longrightarrow {<}x,v{>}{\in}$pending_ts_assign in state s_{i-1}
- ${<}x,v{>}{\in}$pending_ts_assign in state s_{i-1}
  $\Longrightarrow$ status$_x$=UNFL in state s_{i-1} — by Invariant 3.9
- lm_wait(z) in state s_{i-1}
  $\Longrightarrow$ status$_z$=RECV in state s_{i-1} — by definition of lm_wait(z)
- ((status$_x$=UNFL) $\wedge$ (status$_z$=RECV)) in state $s_{i-1} \Longrightarrow x{\neq}z$
- (lm_wait(z) in state s_{i-1}) $\wedge$ ($\pi_i{=}{<}\mathtt{ASSIGN}_x,v{>}$) $\wedge$ ($x{\neq}z$)
  $\Longrightarrow$ lm_wait(z) in state s_i

  and thus the lemma has been proven. □

Lemma A.22

$$\begin{array}{rl} & (s_{i-1} \text{ is a reachable state of } \mathtt{LM}) \\ \wedge & (\neg\text{recvbl}(z) \text{ in state } s_{i-1}) \\ \wedge & (\pi_i{=}{<}\mathtt{ASSIGN}_x,v{>}) \\ \Longrightarrow & \\ & \neg\text{recvbl}(z) \text{ in state } s_i \end{array}$$

Proof:

- $\pi_i{=}{<}\mathtt{ASSIGN}_x,v{>} \Longrightarrow {<}x,v{>}{\in}$pending_ts_assign in state s_{i-1}
- ${<}x,v{>}{\in}$pending_ts_assign in state s_{i-1}
  $\Longrightarrow$ recvbl(x) in state s_{i-1} — by definition of recvbl(x)
- (($\neg$recvbl(z)) $\wedge$ (recvbl(x))) in state $s_{i-1} \Longrightarrow x{\neq}z$
- ($\neg$recvbl(z) in state s_{i-1}) $\wedge$ ($\pi_i{=}{<}\mathtt{ASSIGN}_x,v{>}$) $\wedge$ ($x{\neq}z$)
  $\Longrightarrow$ $\neg$recvbl(z) in state s_i

  and thus the lemma has been proven. □

Theorem A.23

$$\begin{array}{rl} & (s_{i-1} \text{ is a reachable state of } \mathtt{LM}) \\ \wedge & (\pi_i{=}{<}\mathtt{ASSIGN}_x,v{>}) \\ \wedge & ((s_{i-1},{<}\mathtt{ASSIGN}_x,v{>},s_i){\in}\text{steps}(\mathtt{LM})) \\ \wedge & (t'{\in}f(s_{i-1})) \\ \Longrightarrow & \\ & t'{\in}f(s_i) \end{array}$$

Proof:

- $\forall z,\ z \in \mathcal{N}$, (lm_keep($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ keep=z in state t' — by Lemma A.2
- $\forall z,\ z \in \mathcal{N}$, (lm_keep($z$) in state s_{i-1}) $\wedge$ (π_i=<$\texttt{ASSIGN}_x$,v>)
  $\Longrightarrow$ lm_keep(z) in state s_i — by Lemma A.19
- $\forall z,\ z \in \mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$let_erase in state t' — by Lemma A.3
- $\forall z,\ z \in \mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (π_i=<$\texttt{ASSIGN}_x$,v>)
  $\Longrightarrow$ lm_let(z) in state s_i — by Lemma A.20
- $\forall z,\ z \in \mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$wait_erase in state t' — by Lemma A.4
- $\forall z,\ z \in \mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (π_i=<$\texttt{ASSIGN}_x$,v>)
  $\Longrightarrow$ lm_wait(z) in state s_i — by Lemma A.21
- $\forall z,\ z \in \mathcal{N}$, ($\neg$recvbl($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ ((keep$\neq z$) $\wedge$ ($z\notin$let_erase) $\wedge$ ($z\in$wait_erase)) in state t'
  by Lemma A.5
- $\forall z,\ z \in \mathcal{N}$, ($\neg$recvbl($z$) in state s_{i-1}) $\wedge$ (π_i=<$\texttt{ASSIGN}_x$,v>)
  $\Longrightarrow$ $\neg$recvbl(z) in state s_i — by Lemma A.22
- Therefore, from the above deductions it follows that
  t'$\in$f(s_i) — by Definition 3.1
  and thus the theorem has been proven. $\Box$

Theorem A.24

$$\begin{array}{ll} & (s_{i-1} \text{ is a reachable state of } \texttt{LM}) \\ \wedge & (\pi_i = \texttt{CS\_REQD}_x) \\ \wedge & ((s_{i-1}, \texttt{CS\_REQD}_x, s_i) \in \text{steps}(\texttt{LM})) \\ \wedge & (t' \in f(s_{i-1})) \\ \Longrightarrow & \\ & t' \in f(s_i) \end{array}$$

Proof:

- $\forall z,\ z \in \mathcal{N}$, (lm_keep($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ keep=z in state t' — by Lemma A.2
- $\forall z,\ z \in \mathcal{N}$, (lm_keep($z$) in state s_{i-1}) $\wedge$ (π_i=$\texttt{CS\_REQD}_x$)
  $\Longrightarrow$ lm_keep(z) in state s_i
- $\forall z,\ z \in \mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$let_erase in state t' — by Lemma A.3
- $\forall z,\ z \in \mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (π_i=$\texttt{CS\_REQD}_x$)
  $\Longrightarrow$ lm_let(z) in state s_i
- $\forall z,\ z \in \mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$wait_erase in state t' — by Lemma A.4
- $\forall z,\ z \in \mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (π_i=$\texttt{CS\_REQD}_x$)
  $\Longrightarrow$ lm_wait(z) in state s_i

- $\forall z,\ z \in \mathcal{N}$, ($\neg$recvbl($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ ((keep$\neq z$) $\wedge$ ($z \notin$let_erase) $\wedge$ ($z \in$wait_erase)) in state t'
  by Lemma A.5
- $\forall z,\ z \in \mathcal{N}$,
  ($\neg$recvbl(z) in state s_{i-1}) $\wedge$ (π_i=`CS_REQD`$_x$) $\Longrightarrow$ $\neg$recvbl(z) in state s_i
- Therefore, from the above deductions it follows that
  t'$\in$f(s_i) by Definition 3.1
  and thus the theorem has been proven. □

Lemma A.25 (s_{i-1} is a reachable state of `LM`)
$\wedge$ (lm_let(z) in state s_{i-1})
$\wedge$ (π_i=`CS_RECV`$_x$)
$\Longrightarrow$
lm_let(z) in state s_i

Proof:

- π_i=`CS_RECV`$_x$ $\Longrightarrow$ $x \in$send_cs_recv in state s_{i-1}
- $x \in$send_cs_recv in state s_{i-1}
  $\Longrightarrow$ ((curr_reqd_dlr$\neq x$) $\wedge$ (recvbl(x))) in state s_{i-1}
  by Invariants 3.11, 3.7 and 3.15
- Either

  (1) $x=z$
  - (((curr_reqd_dlr$\neq x$) $\wedge$ (recvbl(x))) in state s_{i-1})
    $\wedge$ (π_i=`CS_RECV`$_x$)
    $\wedge$ ($x=z$)
    $\Longrightarrow$ ((curr_reqd_dlr$\neq z$) $\wedge$ (recvbl(z))) in state s_i
  - (π_i=`CS_RECV`$_x$) $\wedge$ ($x=z$) $\Longrightarrow$ pending_ack$_z$=T in state s_i
  - ((curr_reqd_dlr$\neq z$) $\wedge$ (recvbl(z)) $\wedge$ (pending_ack$_z$=T)) in state s_i
    $\Longrightarrow$ lm_let(z) in state s_i

  or

  (2) $x \neq z$
  - (((lm_let(z)) $\wedge$ (recvbl(x))) in state s_{i-1}) $\wedge$ ($x \neq z$)
    $\Longrightarrow$ ((curr_reqd_dlr$\neq z$) $\wedge$ (recvbl(z)) $\wedge$ ($\neg$lm_wait(z)))
    in state s_{i-1}
  - (((curr_reqd_dlr$\neq z$)
    $\wedge$ (recvbl(z))
    $\wedge$ ($\neg$lm_wait(z))) in state s_{i-1})
    $\wedge$ (π_i=`CS_RECV`$_x$)
    $\wedge$ ($x \neq z$)
    $\Longrightarrow$ ((curr_reqd_dlr$\neq z$) $\wedge$ (recvbl(z)) $\wedge$ ($\neg$lm_wait(z)))
    in state s_i
  - ((curr_reqd_dlr$\neq z$) $\wedge$ (recvbl(z)) $\wedge$ ($\neg$lm_wait(z))) in state s_i
    $\Longrightarrow$ lm_let(z) in state s_i by definition of lm_let(z)

- For both possible cases, the desired result is obtained and thus the lemma has been proven. $\square$

Lemma A.26
$$
\begin{array}{ll}
& (\mathrm{s}_{i-1} \text{ is a reachable state of } \mathtt{LM}) \\
\wedge & (\mathrm{lm\_wait}(z) \text{ in state } \mathrm{s}_{i-1}) \\
\wedge & (\pi_i = \mathtt{CS\_RECV}_x) \\
\Longrightarrow & \\
& \mathrm{lm\_wait}(z) \text{ in state } \mathrm{s}_i
\end{array}
$$

Proof:

- $\pi_i = \mathtt{CS\_RECV}_x \Longrightarrow x \in \mathrm{send\_cs\_recv}$ in state s_{i-1}
- $x \in \mathrm{send\_cs\_recv}$ in state s_{i-1}
  $\Longrightarrow$ $\mathrm{status}_x \neq \mathrm{RECV}$ in state s_{i-1} by Invariant 3.15
- ($\mathrm{lm\_wait}(z)$ in state s_{i-1}) $\wedge$ ($\mathrm{status}_x \neq \mathrm{RECV}$ in state s_{i-1})
  $\Longrightarrow z \neq x$ by definition of $\mathrm{lm\_wait}(z)$
- ($\mathrm{lm\_wait}(z)$ in state s_{i-1}) $\wedge$ ($\pi_i = \mathtt{CS\_RECV}_x$) $\wedge$ ($z \neq x$)
  $\Longrightarrow$ $\mathrm{lm\_wait}(z)$ in state s_i

  and thus the lemma has been proven. $\square$

Lemma A.27
$$
\begin{array}{ll}
& (\mathrm{s}_{i-1} \text{ is a reachable state of } \mathtt{LM}) \\
\wedge & (\neg\mathrm{recvbl}(z) \text{ in state } \mathrm{s}_{i-1}) \\
\wedge & (\pi_i = \mathtt{CS\_RECV}_x) \\
\Longrightarrow & \\
& \neg\mathrm{recvbl}(z) \text{ in state } \mathrm{s}_i
\end{array}
$$

Proof:

- $\pi_i = \mathtt{CS\_RECV}_x \Longrightarrow x \in \mathrm{send\_cs\_recv}$ in state s_{i-1}
- $x \in \mathrm{send\_cs\_recv}$ in state s_{i-1}
  $\Longrightarrow$ $\mathrm{recvbl}(x)$ in state s_{i-1} by Invariants 3.7 and 3.15
- (($\neg\mathrm{recvbl}(z)$) $\wedge$ ($\mathrm{recvbl}(x)$)) in state $\mathrm{s}_{i-1} \Longrightarrow z \neq x$
- ($\neg\mathrm{recvbl}(z)$ in state s_{i-1}) $\wedge$ ($\pi_i = \mathtt{CS\_RECV}_x$) $\wedge$ ($z \neq x$)
  $\Longrightarrow$ $\neg\mathrm{recvbl}(z)$ in state s_i

  and thus the lemma has been proven. $\square$

Theorem A.28
$$
\begin{array}{ll}
& (\mathrm{s}_{i-1} \text{ is a reachable state of } \mathtt{LM}) \\
\wedge & (\pi_i = \mathtt{CS\_RECV}_x) \\
\wedge & ((\mathrm{s}_{i-1}, \mathtt{CS\_RECV}_x, \mathrm{s}_i) \in \mathrm{steps}(\mathtt{LM})) \\
\wedge & (\mathrm{t'} \in \mathrm{f}(\mathrm{s}_{i-1})) \\
\Longrightarrow & \\
& \mathrm{t'} \in \mathrm{f}(\mathrm{s}_i)
\end{array}
$$

Proof:

- $\forall z,\ z\in\mathcal{N}$, (lm_keep($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ keep$=z$ in state t' by Lemma A.2
- $\forall z,\ z\in\mathcal{N}$,
  (lm_keep(z) in state s_{i-1}) $\wedge$ (π_i=`CS_RECV`$_x$) $\Longrightarrow$ lm_keep(z) in state s_i
- $\forall z,\ z\in\mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$let_erase in state t' by Lemma A.3
- $\forall z,\ z\in\mathcal{N}$, (lm_let($z$) in state s_{i-1}) $\wedge$ (π_i=`CS_RECV`$_x$)
  $\Longrightarrow$ lm_let(z) in state s_i by Lemma A.25
- $\forall z,\ z\in\mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ $z\in$wait_erase in state t' by Lemma A.4
- $\forall z,\ z\in\mathcal{N}$, (lm_wait($z$) in state s_{i-1}) $\wedge$ (π_i=`CS_RECV`$_x$)
  $\Longrightarrow$ lm_wait(z) in state s_i by Lemma A.26
- $\forall z,\ z\in\mathcal{N}$, ($\neg$recvbl($z$) in state s_{i-1}) $\wedge$ (t'$\in$f(s_{i-1}))
  $\Longrightarrow$ ((keep$\neq z$) $\wedge$ ($z\notin$let_erase) $\wedge$ ($z\in$wait_erase)) in state t'
  by Lemma A.5
- $\forall z,\ z\in\mathcal{N}$, ($\neg$recvbl($z$) in state s_{i-1}) $\wedge$ (π_i=`CS_RECV`$_x$)
  $\Longrightarrow$ $\neg$recvbl(z) in state s_i by Lemma A.27
- Therefore, from the above deductions it follows that
  t'$\in$f(s_i) by Definition 3.1

  and thus the theorem has been proven. $\square$

A.2 Proof of Liveness

Theorem A.69, which is found at the end of this section, states XEL's important liveness property: the DLR for every committed update is eventually erased. Some preliminary lemmas must first be proven which will ultimately contribute toward the proof of Theorem A.69. In all the following lemmas and theorems, let α denote an execution for the `LM` module, and let π_i represent the i^{th} action of α (where $i\in\mathcal{N}$ and $i\geq 1$).

Lemma A.29 π_h=`ACK_ASSIGN`$_x$
$\Longrightarrow$
$\exists g,\ g<h$, s.t. π_g=<`ASSIGN`$_x$,t> for some t

Proof:

- π_h=`ACK_ASSIGN`$_x$ $\Longrightarrow$ ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_{h-1}
- ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_{h-1}
  $\Longrightarrow$ $\exists g,\ g\leq h-1$, s.t. π_g=<`ASSIGN`$_x$,t> for some t

  and thus the lemma has been proven. $\square$

Lemma A.30 $\quad (\alpha \text{ is a well-formed execution})$

$$\begin{aligned} & (\alpha \text{ is a well-formed execution}) \\ \wedge\ & (\pi_i {=} \texttt{<ASSIGN}_x, u\texttt{>} \text{ for some } u) \\ \Longrightarrow\ & \\ & \nexists h,\ h{\neq}i, \text{ s.t. } \pi_h{=}\texttt{<ASSIGN}_x, v\texttt{>} \text{ for any } v \end{aligned}$$

Proof:

By contradiction. Without loss of generality, assume

$\exists h,\ h{<}i$, s.t. $\pi_h{=}\texttt{<ASSIGN}_x,v\texttt{>}$ for some v.

Case 1: $u{=}v$.

- $\pi_h{=}\texttt{<ASSIGN}_x,v\texttt{>} \Longrightarrow \quad (\texttt{<}x,v\texttt{>}{\in}\text{pending\_ts\_assign in state s}_{h-1})$
  $\wedge \quad (\texttt{<}x,v\texttt{>}{\notin}\text{pending\_ts\_assign in state s}_h)$
- $\texttt{<}x,v\texttt{>}{\in}$pending_ts_assign in state $s_{h-1} \Longrightarrow \exists q,\ q{\leq}h{-}1$, s.t. $\pi_q{=}\texttt{COMMIT}_x$
- $(\pi_i{=}\texttt{<ASSIGN}_x,u\texttt{>}) \wedge (u{=}v) \Longrightarrow \texttt{<}x,v\texttt{>}{\in}$pending_ts_assign in state s_{i-1}
- $\quad (\texttt{<}x,v\texttt{>}{\notin}$pending_ts_assign in state $s_h)$
  $\wedge \quad (\texttt{<}x,v\texttt{>}{\in}$pending_ts_assign in state $s_{i-1})$
  $\wedge \quad (h{<}i)$
  $\Longrightarrow \exists r,\ h{<}r{\leq}i{-}1$, s.t. $\pi_r{=}\texttt{COMMIT}_x$
- $\pi_r{=}\texttt{COMMIT}_x \Longrightarrow \nexists q,\ q{\neq}r$, s.t. $\pi_q{=}\texttt{COMMIT}_x$ by WF1
  But this contradicts the earlier deduction that
  $\exists q,\ q{\leq}h{-}1{<}r$, s.t. $\pi_q{=}\texttt{COMMIT}_x$
  and so this case is impossible.

Case 2: $u{\neq}v$.

- $\pi_i{=}\texttt{<ASSIGN}_x,u\texttt{>} \Longrightarrow \texttt{<}x,u\texttt{>}{\in}$pending_ts_assign in state s_{i-1}
- $\texttt{<}x,u\texttt{>}{\in}$pending_ts_assign in state s_{i-1}
  $\Longrightarrow \exists r,\ r{\leq}i{-}1$, s.t. $((\pi_r{=}\texttt{COMMIT}_x) \wedge (u{=}$current_ts in state $s_{r-1}))$
- $\pi_r{=}\texttt{COMMIT}_x \Longrightarrow \nexists q,\ q{\neq}r$, s.t. $\pi_q{=}\texttt{COMMIT}_x$ by WF1
- $\pi_h{=}\texttt{<ASSIGN}_x,v\texttt{>} \Longrightarrow \texttt{<}x,v\texttt{>}{\in}$pending_ts_assign in state s_{h-1}
- $\texttt{<}x,v\texttt{>}{\in}$pending_ts_assign in state s_{h-1}
  $\Longrightarrow \exists q,\ q{\leq}h{-}1$, s.t. $((\pi_q{=}\texttt{COMMIT}_x) \wedge (v{=}$current_ts in state $s_{q-1}))$
- $u{\neq}v \Longrightarrow s_{q-1}{\neq}s_{r-1}$
- $s_{q-1}{\neq}s_{r-1} \Longrightarrow q{\neq}r$
  But this contradicts the earlier deduction that
  $\nexists q,\ q{\neq}r$, s.t. $\pi_q{=}\texttt{COMMIT}_x$
  and so this case is also impossible.

Since both cases are impossible, the original assumption must be false and the lemma has been proven. □

Lemma A.31

$$\begin{aligned} & (\alpha \text{ is a well-formed execution}) \\ \wedge\ & (((\text{status}_x{=}\text{UNFL}) \wedge (\text{pending\_ack}_x{=}\text{T})) \text{ in state s}_i) \\ \Longrightarrow\ & \\ & \nexists h,\ h{\leq}i, \text{ s.t. } \pi_h{=}\texttt{ACK\_ASSIGN}_x \end{aligned}$$

Proof:

By contradiction. Assume $\exists h,\ h \leq i$, s.t. $\pi_h = \texttt{ACK\_ASSIGN}_x$

- ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_i
  $\Longrightarrow \exists g,\ g \leq i$, s.t. $(\pi_g = <\texttt{ASSIGN}_x, t>$ for some $t)$
  $\wedge\ (\not\exists f,\ g \leq f \leq i$, s.t. $\pi_f = \texttt{ACK\_ASSIGN}_x)$
- $\pi_g = <\texttt{ASSIGN}_x, t>$
  $\Longrightarrow \not\exists d,\ d \neq g$, s.t. $\pi_d = <\texttt{ASSIGN}_x, u>$ for any u by Lemma A.30
- $(\exists h,\ h \leq i$, s.t. $\pi_h = \texttt{ACK\_ASSIGN}_x) \wedge (\not\exists f,\ g \leq f \leq i$, s.t. $\pi_f = \texttt{ACK\_ASSIGN}_x)$
  $\Longrightarrow \exists e,\ e < g$, s.t. $\pi_e = \texttt{ACK\_ASSIGN}_x$
- $\pi_e = \texttt{ACK\_ASSIGN}_x$
  $\Longrightarrow \exists d,\ d < e < g$, s.t. $\pi_d = <\texttt{ASSIGN}_x, u>$ for some u by Lemma A.29

But this is a contradiction and so the original assumption must be false. Thus the lemma has been proven. $\square$

Lemma A.32 ((curr_reqd_dlr=x, $x \in \mathcal{N}$) $\wedge$ (curr_reqd_acked=T)) in state s_i
$\Longrightarrow$
$\exists h,\ h \leq i$, s.t. $\pi_h = \texttt{ACK\_ASSIGN}_x$

Proof:

By contradiction. Assume $\not\exists h,\ h \leq i$, s.t. $\pi_h = \texttt{ACK\_ASSIGN}_x$

- curr_reqd_dlr=x, $x \in \mathcal{N}$, in state s_i
  $\Longrightarrow \exists g,\ g \leq i$, s.t. $(\pi_g = \texttt{COMMIT}_x)$
  $\wedge\ (\not\exists f,\ g < f \leq i$, s.t. $\pi_f = \texttt{COMMIT}_y$ for some $y \neq x)$
- $\pi_g = \texttt{COMMIT}_x \Longrightarrow$ ((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=F)) in state s_g
- (((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=F)) in state s_{m-1})
  $\wedge\ (\pi_m \neq \texttt{ACK\_ASSIGN}_x)$
  $\wedge\ (\pi_m \neq \texttt{COMMIT}_y$ for $y \neq x)$
  $\Longrightarrow$ ((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=F)) in state s_m
- (((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=F)) in state s_g)
  $\wedge\ (\not\exists h,\ h \leq i$, s.t. $\pi_h = \texttt{ACK\_ASSIGN}_x)$
  $\wedge\ (\not\exists f,\ g < f \leq i$, s.t. $\pi_f = \texttt{COMMIT}_y$ for $y \neq x)$
  $\Longrightarrow$ ((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=F)) in state s_i
  by induction.

But this contradiction implies that the original assumption must be false and thus the lemma has been proven. $\square$

Lemma A.33 $x \in$send_cs_recv in state s_i
$\Longrightarrow$
$\exists h,\ h \leq i$, s.t. $\pi_h = \texttt{ACK\_ASSIGN}_x$

Proof:

By contradiction. Assume $\nexists h$, $h \leq i$, s.t. π_h=$\texttt{ACK\_ASSIGN}_x$

- $x \in$send_cs_recv in state $s_i \implies$ either

  (1) $\exists g$, $g \leq i$, s.t. $(\pi_g = \texttt{COMMIT}_y$ for some $y)$
  $\wedge$ (curr_reqd_dlr=x in state s_{g-1})
  $\wedge$ (curr_reqd_acked=T in state s_{g-1})

  - ((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=T)) in state s_{g-1}
    $\implies \exists h$, $h \leq g-1$, s.t. π_h=$\texttt{ACK\_ASSIGN}_x$ by Lemma A.32
    But this is a contradiction, so this case cannot be true.

  or

  (2) $\exists h$, $h \leq i$, s.t. π_h=$\texttt{ACK\_ASSIGN}_x$

  - But this is a contradiction, so this case cannot be true.

  or

  (3) $\exists g$, $g \leq i$, s.t. $(\pi_g = \texttt{<DLR\_GONE},t\texttt{>}$ for some $t)$
  $\wedge$ (curr_reqd_dlr=x in state s_{g-1})
  $\wedge$ (curr_reqd_acked=T) in state s_{g-1})

  - ((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=T)) in state s_{g-1}
    $\implies \exists h$, $h \leq g-1$, s.t. π_h=$\texttt{ACK\_ASSIGN}_x$ by Lemma A.32
    But this is a contradiction, so this case cannot be true.

- Since all three possible cases lead to contradictions, the original assumption must be false and thus the lemma has been proven. □

Lemma A.34

$$
\begin{aligned}
& (\alpha \text{ is a well-formed execution}) \\
\wedge\ & (\pi_j = \texttt{ERASE}_x) \\
\implies & \\
& \exists f,\ f<j, \text{ s.t. } \pi_f = \texttt{CS\_RECV}_x
\end{aligned}
$$

Proof:

- π_j=$\texttt{ERASE}_x \implies \exists h$, $h<j$, s.t. π_h=$\texttt{ERASABLE}_x$ by WF2
- π_h=$\texttt{ERASABLE}_x \implies x \in$pending_erasable in state s_{h-1}
- $x \in$pending_erasable in state $s_{h-1} \implies \exists g$, $g \leq h-1$, s.t. π_g=$\texttt{ACK\_CS\_RECV}_x$
- π_g=$\texttt{ACK\_CS\_RECV}_x \implies$ status$_x$=RECV in state s_{g-1}
- status$_x$=RECV in state $s_{g-1} \implies \exists f$, $f \leq g-1$, s.t. π_f=$\texttt{CS\_RECV}_x$
  and thus the lemma has been proven. □

Lemma A.35

$$
\begin{aligned}
& (\text{curr\_reqd\_dlr}=x \text{ in state } s_i \text{ for some } x \in \mathcal{N}) \\
\wedge\ & (\nexists h,\ h \leq i, \text{ s.t. } \pi_h = \texttt{ACK\_ASSIGN}_x) \\
\implies & \\
& \text{curr\_reqd\_acked=F in state } s_i
\end{aligned}
$$

Proof:

By contradiction. Assume curr_reqd_acked=T in state s_i

- curr_reqd_acked=T in state s_i
  $\Longrightarrow \exists h,\ h \le i$, s.t. $(\pi_h = \mathtt{ACK\_ASSIGN}_y)$
  $\wedge$ (curr_reqd_dlr=y, for some $y \in \mathcal{N}$, in state s_{h-1})
  $\wedge$ $(\not\exists j,\ h < j \le i$, s.t. $\pi_j = \mathtt{COMMIT}_z$ for some $z \in \mathcal{N})$
- (curr_reqd_dlr=y, $y \in \mathcal{N}$, in state s_{h-1}) $\wedge$ $(\pi_h = \mathtt{ACK\_ASSIGN}_y)$
  $\Longrightarrow$ ((curr_reqd_dlr=y) $\vee$ (curr_reqd_dlr=$\perp$)) in state s_h
- (((curr_reqd_dlr=y, $y \in \mathcal{N}$) $\vee$ (curr_reqd_dlr=$\perp$)) in state s_h)
  $\wedge$ $(\not\exists j,\ h < j \le i$, s.t. $\pi_j = \mathtt{COMMIT}_z$ for some $z \in \mathcal{N})$
  $\Longrightarrow \forall l,\ h \le l \le i$, ((curr_reqd_dlr=$y$, $y \in \mathcal{N}$) $\vee$ (curr_reqd_dlr=$\perp$)) in state s_l
  by definition of steps(LOT)
- Either

  (1) curr_reqd_dlr=y, $y \in \mathcal{N}$, in state s_i

  - By transitivity, $x = y$ so $\exists h,\ h \le i$, s.t. $\pi_h = \mathtt{ACK\_ASSIGN}_x$.
    But this contradicts the lemma's predicate.

  or

  (2) curr_reqd_dlr=$\perp$ in state s_i

  - But this also contradicts the lemma's predicate.

  Since both possible cases lead to contradictions, the original assumption must be false and so the lemma has been proven. □

Lemma A.36 (α is a well-formed execution)
$\wedge$ $(\pi_i = \mathtt{ERASE}_x)$
$\Longrightarrow$
$\exists h,\ h < i$, s.t. $\pi_h = \mathtt{ACK\_ASSIGN}_x$

Proof:

By contradiction. Assume $\not\exists h,\ h < i$, s.t. $\pi_h = \mathtt{ACK\_ASSIGN}_x$

- $\pi_i = \mathtt{ERASE}_x \Longrightarrow \exists e,\ e < i$, s.t. $\pi_e = \mathtt{CS\_RECV}_x$ by Lemma A.34
- $\pi_e = \mathtt{CS_RECV}_x \Longrightarrow x \in$send_cs_recv in state s_{e-1}
- $x \in$send_cs_recv in state $s_{e-1} \Longrightarrow \exists d,\ d \le e-1$, s.t. either

  (1) ($(\pi_d = \mathtt{COMMIT}_y$ for some $y \in \mathcal{N})$
  $\wedge$ (((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=T)) in state s_{d-1}))

  - (curr_reqd_dlr=x in state s_{d-1})
    $\wedge$ $(\not\exists h,\ h \le d-1 \le i$, s.t $\pi_h = \mathtt{ACK\_ASSIGN}_x)$
    $\Longrightarrow$ curr_reqd_acked=F in state s_{d-1} by Lemma A.35
    But this is a contradiction, and so this case could not possibly occur.

  or

  (2) $\pi_d = \mathtt{ACK\_ASSIGN}_x$

  - But this contradicts the assumption, and so this case can never occur.

  or

(3) ($(\pi_d = \texttt{<DLR\_GONE},ts_y\texttt{>}$ for some $ts_y)$
$\wedge$ (curr_reqd_dlr=x in state s_{d-1})
$\wedge$ (curr_reqd_acked=T in state s_{d-1}))

- (curr_reqd_dlr=x in state s_{d-1})
  $\wedge$ ($\nexists h$, $h \leq d-1 \leq i$, s.t. $\pi_h = \texttt{ACK\_ASSIGN}_x$)
  $\Longrightarrow$ curr_reqd_acked=F in state s_{d-1} by Lemma A.35

  But this is a contradiction and so this case also cannot occur.

Since none of these three cases can be true and there are no other possibilities, the original assumption must be false and so the lemma has been proven. □

Lemma A.37 (α is a well-formed execution)
$\wedge$ (((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_k)
$\wedge$ ($\exists q$, $q>k$, s.t. $\nexists n$, $n<q$, s.t. $\pi_n = \texttt{ACK\_ASSIGN}_x$)
$\Longrightarrow$
$\nexists p$, $p<q$, s.t. $(\pi_p = \texttt{CS\_REQD}_x) \vee (\pi_p = \texttt{CS\_RECV}_x) \vee (\pi_p = \texttt{ERASE}_x)$

Proof:

By contradiction.

Assume

$\exists p$, $p<q$, s.t. $(\pi_p = \texttt{CS\_REQD}_x) \vee (\pi_p = \texttt{CS\_RECV}_x) \vee (\pi_p = \texttt{ERASE}_x)$

- Either

  (1) $\exists p$, $p<q$, s.t. $\pi_p = \texttt{CS\_REQD}_x$
  - $\pi_p = \texttt{CS\_REQD}_x \Longrightarrow x$=send_cs_reqd in state s_{p-1}
  - x=send_cs_reqd in state $s_{p-1} \Longrightarrow \exists n$, $n \leq p-1$, s.t. $\pi_n = \texttt{ACK\_ASSIGN}_x$
    But this contradicts the lemma's predicate and so this case cannot occur.

  or

  (2) $\exists p$, $p<q$, s.t. $\pi_p = \texttt{CS\_RECV}_x$
  - $\pi_p = \texttt{CS_RECV}_x \Longrightarrow x \in$send_cs_recv in state s_{p-1}
  - $x \in$send_cs_recv in state $s_{p-1} \Longrightarrow$ either

    (i) $\exists m$, $m \leq p-1$, s.t. ($\pi_m = \texttt{COMMIT}_y$ for some $y \in \mathcal{N}$)
    $\wedge$ (curr_reqd_dlr=x in state s_{m-1})
    $\wedge$ (curr_reqd_acked=T in state s_{m-1})
    - ((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=T)) in state s_{m-1}
      $\Longrightarrow \exists n$, $n \leq m-1$, s.t. $\pi_n = \texttt{ACK\_ASSIGN}_x$
      by Lemma A.32

      But this contradicts the lemma's predicate and so this subcase cannot occur.

    or

    (ii) $\exists n$, $n \leq p-1$, s.t. $\pi_n = \texttt{ACK\_ASSIGN}_x$
    - But this contradicts the lemma's predicate, and so this subcase cannot occur either.

or

(iii) $\exists m,\ m \leq p-1$, s.t. $(\pi_m = \text{<}\texttt{DLR\_GONE},t\text{>}$ for some $t)$
$\wedge$ (curr_reqd_dlr=x in state s_{m-1})
$\wedge$ (curr_reqd_acked=T in state s_{m-1})

- ((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=T)) in state s_{m-1}
  $\Longrightarrow \exists n,\ n \leq m-1$, s.t. $\pi_n = \texttt{ACK\_ASSIGN}_x$
  by Lemma A.32

  But this contradicts the lemma's predicate and so this subcase cannot occur either.

Since all three subcases lead to contradictions and there are no other possible subcases, the entire case (2) must be impossible.

or

(3) $\exists p,\ p<q$, s.t. $\pi_p = \texttt{ERASE}_x$

- $\pi_p = \texttt{ERASE}_x$
  $\Longrightarrow \exists n,\ n<p$, s.t. $\pi_n = \texttt{ACK\_ASSIGN}_x$ by Lemma A.36

  But this contradicts the lemma's predicate, and so also this case must be impossible.

- All possible cases lead to contradictions. Therefore, the original assumption must be false and thus the lemma has been proven. □

Lemma A.38

$$\pi_i = \texttt{ACK\_ASSIGN}_x \\ \Longrightarrow \\ \exists h,\ h<i, \text{ s.t. } \pi_h = \texttt{COMMIT}_x$$

Proof:

- $\pi_i = \texttt{ACK\_ASSIGN}_x \Longrightarrow$ ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_{i-1}
- ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_{i-1}
  $\Longrightarrow \exists j,\ j \leq i-1$, s.t. $\pi_j = \text{<}\texttt{ASSIGN}_x,ts_x\text{>}$ for some ts_x
- $\pi_j = \text{<}\texttt{ASSIGN}_x,ts_x\text{>} \Longrightarrow \text{<}x,ts_x\text{>} \in$ pending_ts_assign in state s_{j-1}
- $\text{<}x,ts_x\text{>} \in$ pending_ts_assign in state $s_{j-1} \Longrightarrow \exists h,\ h \leq j-1$, s.t. $\pi_h = \texttt{COMMIT}_x$
- By transitivity,
  $\exists h,\ h<i$, s.t. $\pi_h = \texttt{COMMIT}_x$
  and thus the lemma has been proven. □

Lemma A.39

$$(\alpha \text{ is a well-formed execution}) \\ \wedge\ (\pi_g = \texttt{COMMIT}_x) \\ \Longrightarrow \\ \forall f,\ f \leq g,\ x \notin \text{send\_cs\_recv in state } s_f$$

Proof:

By contradiction. Assume $\exists f,\ f \leq g$, s.t. $x \in$ send_cs_recv in state s_f

- π_g=$\texttt{COMMIT}_x \implies \nexists d$, $d<g$, s.t. π_d=$\texttt{COMMIT}_x$ by WF1
- $x\in$send_cs_recv in state $s_f \implies \exists e$, $e\leq f$, s.t. π_e=$\texttt{ACK\_ASSIGN}_x$ by Lemma A.33
- π_e=$\texttt{ACK\_ASSIGN}_x \implies \exists d$, $d<e$, s.t. π_d=$\texttt{COMMIT}_x$ by Lemma A.38
- $(d<e\leq f\leq g) \wedge (\exists d, d<e$, s.t. π_d=$\texttt{COMMIT}_x) \implies \exists d$, $d<g$, s.t. π_d=$\texttt{COMMIT}_x$
  But this is a contradiction, and so the original assumption must be false. Thus the lemma has been proven. □

Lemma A.40

$$\begin{aligned} & (\alpha \text{ is a well-formed execution}) \\ \wedge \quad & (x\in\text{send\_cs\_recv in state } s_i) \\ \implies & \\ & \text{curr\_reqd\_dlr}\neq x \text{ in state } s_i \end{aligned}$$

Proof:

By induction.

- $x\in$send_cs_recv in state $s_i \implies \exists h$, $h\leq i$, s.t. π_h=$\texttt{ACK\_ASSIGN}_x$ by Lemma A.33
- π_h=$\texttt{ACK\_ASSIGN}_x \implies \exists g$, $g<h$, s.t. π_g=$\texttt{COMMIT}_x$ by Lemma A.38
- π_g=$\texttt{COMMIT}_x \implies \nexists m$, $m>g$, s.t. π_m=$\texttt{COMMIT}_x$ by WF1
- π_g=$\texttt{COMMIT}_x \implies$ curr_reqd_dlr=x in state s_g
- π_g=$\texttt{COMMIT}_x \implies \forall f$, $f\leq g$, $x\notin$send_cs_recv in state s_f by Lemma A.39
- $\forall f$, $f\leq g$, $x\notin$send_cs_recv in state s_f
  $\implies \forall f$, $f\leq g$, ((curr_reqd_dlr$\neq x$) $\vee$ ($x\notin$send_cs_recv)) in state s_f
- (((curr_reqd_dlr$\neq x$) $\vee$ ($x\notin$send_cs_recv)) in state s_{m-1}) $\wedge$ ($\pi_m\neq\texttt{COMMIT}_x$)
  $\implies$ ((curr_reqd_dlr$\neq x$) $\vee$ ($x\notin$send_cs_recv)) in state s_m
  by definition of steps($\texttt{LOT}$)
- Therefore, by induction,
  $\forall p$, $p>g$, ((curr_reqd_dlr$\neq x$) $\vee$ ($x\notin$send_cs_recv)) in state s_p
- Hence,
  $\forall q$, $q>0$, ((curr_reqd_dlr$\neq x$) $\vee$ ($x\notin$send_cs_recv)) in state s_q
- (($x\in$send_cs_recv) $\wedge$ ((curr_reqd_dlr$\neq x$) $\vee$ ($x\notin$send_cs_recv))) in state s_i
  $\implies$ curr_reqd_dlr$\neq x$ in state s_i
  and thus the lemma has been proven. □

Lemma A.41

$$\begin{aligned} & (\alpha \text{ is a well-formed execution}) \\ \wedge \quad & (\pi_i = \texttt{ACK\_ASSIGN}_x) \\ \implies & \\ & \nexists h,\ h\neq i, \text{ s.t. } \pi_h = \texttt{ACK\_ASSIGN}_x \end{aligned}$$

Proof:

By contradiction.
Without loss of generality, assume $\exists h$, $h<i$, s.t. π_h=$\texttt{ACK\_ASSIGN}_x$

- π_h=$\texttt{ACK\_ASSIGN}_x \implies$ ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_{h-1}
- ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_{h-1}
  $\implies \exists g$, $g \leq h-1$, s.t π_g=<$\texttt{ASSIGN}_x$,ts_x> for some ts_x
- π_h=$\texttt{ACK\_ASSIGN}_x \implies$ ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=F)) in state s_h
- π_i=$\texttt{ACK\_ASSIGN}_x \implies$ ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_{i-1}
- (((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=F)) in state s_h)
  $\wedge$ (((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_{i-1})
  $\wedge$ ($h<i$)
  $\implies \exists f$, $h<f \leq i-1$, s.t. π_f=<$\texttt{ASSIGN}_x$,u> for some u
- π_f=<$\texttt{ASSIGN}_x$,u> $\implies \nexists g$, $g<f$, s.t. π_g=<$\texttt{ASSIGN}_x$,ts_x> for any ts_x
  by Lemma A.30

But this is a contradiction, and so the assumption must be false. Thus the lemma has been proven. □

Lemma A.42 (α is a well-formed execution)
$\wedge$ (π_g=$\texttt{CS\_RECV}_x$)
$\implies$
$\nexists d$, $d \neq g$, s.t. π_d=$\texttt{CS\_RECV}_x$

Proof:

By contradiction. Without loss of generality, assume $\exists d$, $d<g$, s.t. π_d=$\texttt{CS\_RECV}_x$

- π_d=$\texttt{CS\_RECV}_x$
  $\implies$ ($x \in$send_cs_recv in state s_{d-1}) $\wedge$ ($x \notin$send_cs_recv in state s_d)
- $x \in$send_cs_recv in state s_{d-1}
  $\implies \exists c$, $c \leq d-1$, s.t. π_c=$\texttt{ACK\_ASSIGN}_x$ by Lemma A.33
- π_c=$\texttt{ACK\_ASSIGN}_x \implies \nexists q$, $q>c$, s.t. π_q=$\texttt{ACK\_ASSIGN}_x$ by Lemma A.41
- π_c=$\texttt{ACK\_ASSIGN}_x \implies \exists b$, $b<c$, s.t. π_b=$\texttt{COMMIT}_x$ by Lemma A.38
- π_b=$\texttt{COMMIT}_x \implies \nexists a$, $a>b$, s.t. π_a=$\texttt{COMMIT}_x$ by WF1
- $x \in$send_cs_recv in state $s_{d-1} \implies$ curr_reqd_dlr$\neq x$ in state s_{d-1}
  by Lemma A.40
- (curr_reqd_dlr$\neq x$ in state s_{d-1}) $\wedge$ ($\nexists a$, $a>b$, s.t. π_a=$\texttt{COMMIT}_x$) $\wedge$ ($b<c<d$)
  $\implies \forall p$, $p \geq d-1$, curr_reqd_dlr$\neq x$ in state s_p
- π_g=$\texttt{CS_RECV}_x \implies x \in$send_cs_recv in state s_{g-1}
- ($x \notin$send_cs_recv in state s_d)
  $\wedge$ ($x \in$send_cs_recv in state s_{g-1})
  $\wedge$ ($d<g$)
  $\implies$ either
  (1) $\exists q$, $d<q \leq g-1$, s.t. (π_q=$\texttt{COMMIT}_y$ for some y)
  $\wedge$ (curr_reqd_dlr=x in state s_{q-1})
  $\wedge$ (curr_reqd_acked=T in state s_{q-1})

- But curr_reqd_dlr=x in state s_{q-1} is a contradiction, so this case cannot occur.

or

(2) $\exists q,\ d<q\leq g-1$, s.t. π_q=$\texttt{ACK\_ASSIGN}_x$

- But this is a contradiction, so this case cannot occur.

or

(3) $\exists q,\ d<q\leq g-1$, s.t. $\quad (\pi_q$=<$\texttt{DLR\_GONE}$,t> for some t)
$\wedge$ (curr_reqd_dlr=x in state s_{q-1})
$\wedge$ (curr_reqd_acked=T in state s_{q-1})

- But curr_reqd_dlr=x in state s_{q-1} is a contradiction, so this case cannot occur.

- Since all three cases lead to contradictions and there are no other possible cases besides these, the original assumption must be false and thus the lemma has been proven. □

Lemma A.43 $\quad \pi_g$=$\texttt{CS\_RECV}_x$
$\Longrightarrow$
$\exists f,\ f<g$, s.t. π_f=<$\texttt{ASSIGN}_x$,t> for some t

Proof:

- π_g=$\texttt{CS_RECV}_x \Longrightarrow x\in$send_cs_recv in state s_{g-1}
- $x\in$send_cs_recv in state s_{g-1}
  $\Longrightarrow \exists e,\ e\leq g-1$, s.t. π_e=$\texttt{ACK\_ASSIGN}_x$ by Lemma A.33
- π_e=$\texttt{ACK\_ASSIGN}_x$
  $\Longrightarrow \exists f,\ f<e$, s.t. π_f=<$\texttt{ASSIGN}_x$,t> for some t by Lemma A.29

  and thus the lemma has been proven. □

Lemma A.44 $\quad$ (pending_ack$_x$=F in state s_h)
$\wedge$ (((status$_x$=RECV) $\wedge$ (pending_ack$_x$=T)) in state s_i)
$\wedge$ ($h<i$)
$\wedge$ ($\nexists e,\ e>h$, s.t. π_e=<$\texttt{ASSIGN}_x$,u> for any u)
$\Longrightarrow$
$\exists d,\ h<d\leq i$, s.t. π_d=$\texttt{CS\_RECV}_x$

Proof:

By contradiction. Assume $\nexists d,\ h<d\leq i$, s.t. π_d=$\texttt{CS\_RECV}_x$

- (((status$_x\neq$RECV) $\vee$ (pending_ack$_x$=F)) in state s_{m-1})
  $\wedge$ ($\pi_m\neq$<$\texttt{ASSIGN}_x$,u> for any u)
  $\wedge$ ($\pi_m\neq\texttt{CS\_RECV}_x$)
  $\Longrightarrow$ ((status$_x\neq$RECV) $\vee$ (pending_ack$_x$=F)) in state s_m

- $\quad$ (pending_ack$_x$=F in state s$_h$)
  $\wedge$ $(h<i)$
  $\wedge$ ($\nexists e$, $e>h$, s.t. π_e=<$\texttt{ASSIGN}_x$,u> for any u)
  $\wedge$ ($\nexists d$, $h<d\leq i$, s.t. π_d=$\texttt{CS\_RECV}_x$)
  $\Longrightarrow$ ((status$_x\neq$RECV) $\vee$ (pending_ack$_x$=F)) in state s$_i$ by induction

  But this contradicts the lemma's predicate, and so the original assumption must be false. Thus the lemma has been proven. □

Lemma A.45 $\quad$ (α is a well-formed execution)
$\wedge$ (((status$_x$=RECV) $\wedge$ (pending_ack$_x$=T)) in state s$_i$)
$\Longrightarrow$
$\exists h$, $h<i$, s.t. π_h=$\texttt{ACK\_CS\_RECV}_x$

Proof:

By contradiction. Assume $\nexists h$, $h<i$, s.t. π_h=$\texttt{ACK\_CS\_RECV}_x$

- π_h=$\texttt{ACK\_CS\_RECV}_x$
  $\Longrightarrow$ (((status$_x$=RECV) $\wedge$ (pending_ack$_x$=T)) in state s$_{h-1}$)
  $\wedge$ (((status$_x$=RECV) $\wedge$ (pending_ack$_x$=F)) in state s$_h$)
- status$_x$=RECV in state s$_{h-1}$
  $\Longrightarrow$ $\exists g$, $g\leq h-1$, s.t. π_g=$\texttt{CS\_RECV}_x$
- π_g=$\texttt{CS\_RECV}_x$ $\Longrightarrow$ $\exists f$, $f<g$, s.t. π_f=<$\texttt{ASSIGN}_x$,t> for some t
  by Lemma A.43
- π_g=$\texttt{CS\_RECV}_x$ $\Longrightarrow$ $\nexists d$, $d\neq g$, s.t. π_d=$\texttt{CS\_RECV}_x$ by Lemma A.42
- π_f=<$\texttt{ASSIGN}_x$,t>
  $\Longrightarrow$ $\nexists e$, $e>f$, s.t. π_e=<$\texttt{ASSIGN}_x$,u> for any u by Lemma A.30
- $(f<g<h)$ $\wedge$ ($\nexists e$, $e>f$, s.t. π_e=<$\texttt{ASSIGN}_x$,u> for any u)
  $\Longrightarrow$ $\nexists e$, $e>h$, s.t. π_e=<$\texttt{ASSIGN}_x$,u> for any u
- $\quad$ (pending_ack$_x$=F in state s$_h$)
  $\wedge$ (((status$_x$=RECV) $\wedge$ (pending_ack$_x$=T)) in state s$_i$)
  $\wedge$ $(h<i)$
  $\wedge$ ($\nexists e$, $e>h$, s.t. π_e=<$\texttt{ASSIGN}_x$,u> for any u)
  $\Longrightarrow$ $\exists d$, $h<d\leq i$, s.t. π_d=$\texttt{CS\_RECV}_x$ by Lemma A.44

  But this is a contradiction, and so the original assumption must be false. Thus the lemma has been proven. □

Lemma A.46 $\quad$ (α is a well-formed execution)
$\wedge$ (((status$_x$=RECV) $\wedge$ (pending_ack$_x$=T)) in state s$_{m-1}$)
$\Longrightarrow$
$\pi_m\neq\texttt{ERASE}_x$

Proof:

By contradiction. Assume π_m=$\texttt{ERASE}_x$.

- ((status$_x$=RECV) $\wedge$ (pending_ack$_x$=T)) in state s$_{m-1}$
  $\Longrightarrow \nexists h$, $h<m-1$, s.t. π_h=$\texttt{ACK\_CS\_RECV}_x$ by Lemma A.45
- π_m=$\texttt{ERASE}_x \Longrightarrow \exists g$, $g<m$, s.t. π_g=$\texttt{ERASABLE}_x$ by WF2
- π_g=$\texttt{ERASABLE}_x \Longrightarrow x\in$pending_erasable in state s$_{g-1}$
- $x\in$pending_erasable in state s$_{g-1}$
  $\Longrightarrow \exists f$, $f\leq g-1<m-1$, s.t. π_f=$\texttt{ACK\_CS\_RECV}_x$

  This contradiction implies that the original assumption must be false and thus the lemma has been proven. □

Lemma A.47 (α is a well-formed and fair execution)
$\wedge$ (((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s$_k$)
$\Longrightarrow$
$\exists l$, $l>k$, s.t. π_l=$\texttt{ACK\_ASSIGN}_x$

Proof:

- ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s$_k$
  $\Longrightarrow \exists q$, $q>k$, s.t. $\nexists n$, $n<q$, s.t. π_n=$\texttt{ACK\_ASSIGN}_x$ by Lemma A.31
- (((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s$_k$)
  $\wedge$ ($\exists q$, $q>k$, s.t. $\nexists n$, $n<q$, s.t. π_n=$\texttt{ACK\_ASSIGN}_x$)
  $\Longrightarrow \nexists p$, $p<q$, s.t. (π_p=$\texttt{CS\_REQD}_x$) $\vee$ (π_p=$\texttt{CS\_RECV}_x$) $\vee$ (π_p=$\texttt{ERASE}_x$)
  by Lemma A.37
- (((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s$_{m-1}$)
  $\wedge$ ($\pi_m\neq\texttt{CS\_REQD}_x$)
  $\wedge$ ($\pi_m\neq\texttt{CS\_RECV}_x$)
  $\wedge$ ($\pi_m\neq\texttt{ERASE}_x$)
  $\Longrightarrow$ ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s$_m$
- By induction and the definition of a fair execution, it therefore follows that
  $\exists l$, $l>k$, s.t. π_l=$\texttt{ACK\_ASSIGN}_x$

  and thus the lemma has been proven. □

Lemma A.48
(($<x,t>\in$pending_ts_assign) $\vee$ (timestamp$_x$=t, $t\in\mathcal{N}$)) in state s$_i$
$\Longrightarrow$
$\exists h$, $h\leq i$, s.t. (π_h=$\texttt{COMMIT}_x$) $\wedge$ (current_ts=t, $t\in\mathcal{N}$, in state s$_{h-1}$)

Proof:

- Either

  (1) $<x,t>\in$pending_ts_assign in state s$_i$

- $<x,t>\in$pending_ts_assign in state s_i
  $\Longrightarrow \exists h,\ h\leq i$, s.t. $(\pi_h=\texttt{COMMIT}_x)$
  $\wedge$ (current_ts$=t$, $t\in\mathcal{N}$, in state s_{h-1})
  by definition of steps(LOT)

or

(2) timestamp$_x=t$, $t\in\mathcal{N}$, in state s_i

- timestamp$_x=t$, $t\in\mathcal{N}$, in state $s_i \Longrightarrow \exists g,\ g\leq i$, s.t. $\pi_g=<\texttt{ASSIGN}_x,t>$
- $\pi_g=<\texttt{ASSIGN}_x,t> \Longrightarrow <x,t>\in$pending_ts_assign in state s_{g-1}
- $<x,t>\in$pending_ts_assign in state s_{g-1}
  $\Longrightarrow \exists h,\ h\leq g-1\leq i$, s.t.
  $(\pi_h=\texttt{COMMIT}_x)$
  $\wedge$ (current_ts$=t$, $t\in\mathcal{N}$, in state s_{h-1})

and thus the lemma has been proven. □

Lemma A.49 $i\leq j$
$\Longrightarrow$
(current_ts in state s_i)$\leq$(current_ts in state s_j)

Proof:

By induction.

- The basis case of $i=j$ is trivial.
- For any action π_j, either
  (current_ts in state s_{j-1})=(current_ts in state s_j)
  for $\pi_j\neq\texttt{COMMIT}_x$ for any $x\in\mathcal{N}$
  or
  (current_ts in state s_{j-1})+1=(current_ts in state s_j)
  for $\pi_j=\texttt{COMMIT}_x$ for some $x\in\mathcal{N}$

  Therefore, for any action π_j,
  (current_ts in state s_{j-1})$\leq$(current_ts in state s_j)
- By the inductive hypothesis,
  $i\leq j-1 \Longrightarrow$ (current_ts in state s_i)$\leq$(current_ts in state s_{j-1})
- Therefore,
  $(i\leq j-1)\wedge$ ((current_ts in state s_{j-1})$\leq$(current_ts in state s_j))
  $\Longrightarrow$ ((current_ts in state s_i)$\leq$(current_ts in state s_j))

  and so the lemma has been proven. □

Lemma A.50

(α is a well-formed execution)
$\wedge$ (curr_reqd_ts$=t$, $t\in\mathcal{N}$, in state s_i)
$\wedge$ ((($<x,t>\in$pending_ts_assign) $\vee$ (timestamp$_x=t$)) in state s_i)
$\Longrightarrow$
curr_reqd_dlr$=x$ in state s_i

Proof:

- $((<x,t>\in$pending_ts_assign$) \vee ($timestamp$_x=t,\ t\in\mathcal{N}))$ in state s_i
  $\Longrightarrow \exists h,\ h\leq i$, s.t. $(\pi_h=\texttt{COMMIT}_x) \wedge ($current_ts$=t$ in state $s_{h-1})$
  by Lemma A.48
- $(\pi_h=\texttt{COMMIT}_x) \wedge ($current_ts$=t$ in state $s_{h-1})$
  $\Longrightarrow (($curr_reqd_dlr$=x) \wedge ($curr_reqd_ts$=t) \wedge ($current_ts$=t+1))$ in state s_h
- (current_ts$=t+1$ in state s_h)
  $\Longrightarrow \forall j,\ j\geq h,\ t<$current_ts in state s_j by Lemma A.49
- $(\forall j,\ j\geq h,\ t<$current_ts in state $s_j)$
  $\wedge$ (curr_reqd_ts$=t,\ t\in\mathcal{N}$, in state s_i)
  $\wedge$ $(h\leq i)$
  $\Longrightarrow \nexists k,\ h<k\leq i$, s.t. $\pi_k=\texttt{COMMIT}_y$ for any y
- (curr_reqd_dlr$=x$ in state s_h)
  $\wedge$ $(\nexists k,\ h<k\leq i$, s.t. $\pi_k=\texttt{COMMIT}_y$ for any $y)$
  $\wedge$ (curr_reqd_ts$=t,\ t\neq\perp$, in state s_i)
  $\Longrightarrow$ curr_reqd_dlr$=x$ in state s_i by definition of steps(LOT)

and thus the lemma has been proven. □

Lemma A.51 (α is a well-formed execution)
$\wedge$ $(\pi_i=\texttt{ERASE}_x)$
$\Longrightarrow$
$\forall k,\ k\geq i$, curr_reqd_dlr$\neq x$ in state s_k

Proof:

- $\pi_i=\texttt{ERASE}_x \Longrightarrow \exists h,\ h<i$, s.t. $\pi_h=\texttt{CS\_RECV}_x$ by Lemma A.34
- $\pi_h=\texttt{CS_RECV}_x \Longrightarrow x\in$send_cs_recv in state s_{h-1}
- $x\in$send_cs_recv in state $s_{h-1} \Longrightarrow$ curr_reqd_dlr$\neq x$ in state s_{h-1}
  by Lemma A.40
- $\pi_h=\texttt{CS\_RECV}_x \Longrightarrow \exists g,\ g<h$, s.t. $\pi_g=<\texttt{ASSIGN}_x,t>$ for some t
  by Lemma A.43
- $\pi_g=<\texttt{ASSIGN}_x,t> \Longrightarrow <x,t>\in$pending_ts_assign in state s_{g-1}
- $<x,t>\in$pending_ts_assign in state $s_{g-1} \Longrightarrow \exists f,\ f\leq g-1$, s.t. $\pi_f=\texttt{COMMIT}_x$
- $\pi_f=\texttt{COMMIT}_x \Longrightarrow \nexists j,\ j>f$, s.t. $\pi_j=\texttt{COMMIT}_x$ by WF1
- (curr_reqd_dlr$\neq x,\ x\in\mathcal{N}$, in state $s_{m-1}) \wedge (\pi_m\neq\texttt{COMMIT}_x)$
  $\Longrightarrow$ curr_reqd_dlr$\neq x$ in state s_m
- (curr_reqd_dlr$\neq x$ in state s_{h-1})
  $\wedge$ $(\nexists j,\ j>f$, s.t. $\pi_j=\texttt{COMMIT}_x)$
  $\wedge$ $(f\leq g-1<h<i)$
  $\Longrightarrow \forall k,\ k\geq i$, curr_reqd_dlr$\neq x$ in state s_k by induction

and thus the lemma has been proven. □

Lemma A.52 $(\alpha \text{ is a well-formed execution})$

$\wedge \quad (\text{timestamp}_x{=}t,\ t{\in}\mathcal{N},\ \text{in state } s_j)$

$\Longrightarrow$

$\nexists k,\ k{>}j,\ \text{s.t. } \pi_k{=}{<}\texttt{ASSIGN}_x{,}u{>} \text{ for any } u$

Proof:

By contradiction. Assume $\exists k,\ k{>}j$, s.t. $\pi_k{=}{<}\texttt{ASSIGN}_x{,}u{>}$ for some u

- timestamp$_x{=}t$, $t{\in}\mathcal{N}$, in state $s_j \Longrightarrow \exists i,\ i{\leq}j$, s.t. $\pi_i{=}{<}\texttt{ASSIGN}_x{,}t{>}$
- $(i{\leq}j) \wedge (j{<}k) \Longrightarrow i{<}k$
- $\pi_k{=}{<}\texttt{ASSIGN}_x{,}u{>} \Longrightarrow \nexists i,\ i{<}k$, s.t. $\pi_i{=}{<}\texttt{ASSIGN}_x{,}t{>}$ by Lemma A.30

  This contradiction implies that the original assumption must be false, and so the lemma has been proven. □

Lemma A.53 $(\alpha \text{ is a well-formed execution})$

$\wedge \quad (\text{timestamp}_x{=}t,\ t{\in}\mathcal{N},\ \text{in state } s_i)$

$\Longrightarrow$

$\forall l,\ l{\geq}i,\ \text{timestamp}_x{=}t \text{ in state } s_l$

Proof:

- timestamp$_x{=}t$, $t{\in}\mathcal{N}$, in state s_i

  $\Longrightarrow \nexists j,\ j{>}i$, s.t. $\pi_j{=}{<}\texttt{ASSIGN}_x{,}u{>}$ for any u by Lemma A.52
- (timestamp$_x{=}t$ in state s_{m-1}) $\wedge$ ($\pi_m{\neq}{<}\texttt{ASSIGN}_x{,}u{>}$ for any u)

  $\Longrightarrow$ timestamp$_x{=}t$ in state s_m
- (timestamp$_x{=}t$ in state s_i) $\wedge$ ($\nexists j,\ j{>}i$, s.t. $\pi_j{=}{<}\texttt{ASSIGN}_x{,}u{>}$ for any u)

  $\Longrightarrow \forall l,\ l{\geq}i$, timestamp$_x{=}t$ in state s_l by induction

  and thus the lemma has been proven. □

Lemma A.54 $(\alpha \text{ is a well-formed execution})$

$\wedge \quad (\text{timestamp}_x{=}t,\ t{\in}\mathcal{N},\ \text{in state } s_i)$

$\wedge \quad (t{\notin}\text{recv\_tss in state } s_i)$

$\wedge \quad (\exists h,\ h{<}i,\ \text{s.t. } \pi_h{=}\texttt{ERASE}_x)$

$\Longrightarrow$

$\forall j,\ j{\geq}i,\ t{\notin}\text{recv\_tss in state } s_j$

Proof:

By contradiction. Assume $\exists j,\ j{>}i$, $t{\in}$recv_tss in state s_j

- ($t{\notin}$recv_tss in state s_i) $\wedge$ ($t{\in}$recv_tss in state s_j) $\wedge$ $(i{<}j)$

  $\Longrightarrow \exists k,\ i{\leq}k{<}j$, s.t. ($t{\notin}$recv_tss in state s_k) $\wedge$ ($t{\in}$recv_tss in state s_{k+1})
- ($t{\notin}$recv_tss in state s_k) $\wedge$ ($t{\in}$recv_tss in state s_{k+1})

  $\Longrightarrow$ curr_reqd_ts$=t$ in state s_k by definition of steps(`LOT`)

- ($\text{timestamp}_x{=}t$, $t{\in}\mathcal{N}$, in state s_i)
  $\implies \forall l$, $l{\geq}i$, $\text{timestamp}_x{=}t$ in state s_l — by Lemma A.53
- ($\forall l$, $l{\geq}i$, $\text{timestamp}_x{=}t$ in state s_l) $\wedge$ ($k{\geq}i$) $\implies$ $\text{timestamp}_x{=}t$ in state s_k
- ((curr_reqd_ts=t, $t{\in}\mathcal{N}$) $\wedge$ ($\text{timestamp}_x{=}t$)) in state s_k
  $\implies$ curr_reqd_dlr=x in state s_k — by Lemma A.50
- ($\pi_h{=}\texttt{ERASE}_x$) $\wedge$ ($h{<}k$) $\implies$ curr_reqd_dlr$\neq x$ in state s_k — by Lemma A.51

  But this is a contradiction and so the original assumption must be false. Thus the lemma has been proven. □

Lemma A.55 $\quad$ (α is a well-formed and fair execution)
$\wedge$ ($x{\in}$send_cs_recv in state s_i)
$\implies$
$\exists j$, $j{>}i$, s.t. $\pi_j{=}\texttt{ERASE}_x$

Proof:

- $x{\in}$send_cs_recv in state s_i $\implies \exists k$, $k{>}i$, s.t. $\pi_k{=}\texttt{CS\_RECV}_x$
- $\pi_k{=}\texttt{CS\_RECV}_x \implies$ (($\text{status}_x{=}\text{RECV}$) $\wedge$ ($\text{pending\_ack}_x{=}\text{T}$)) in state s_k
- (($\text{status}_x{=}\text{RECV}$) $\wedge$ ($\text{pending\_ack}_x{=}\text{T}$)) in state s_{m-1}
  $\implies \pi_m{\neq}\texttt{ERASE}_x$ — by Lemma A.46
- ((($\text{status}_x{=}\text{RECV}$) $\wedge$ ($\text{pending\_ack}_x{=}\text{T}$)) in state s_{m-1}) $\wedge$ ($\pi_m{\neq}\texttt{ERASE}_x$)
  $\implies$ (($\text{status}_x{=}\text{RECV}$) $\wedge$ ($\text{pending\_ack}_x{=}\text{T}$)) in state s_m
- (($\text{status}_x{=}\text{RECV}$) $\wedge$ ($\text{pending\_ack}_x{=}\text{T}$)) in state s_k
  $\implies \exists n$, $n{>}k$, s.t. $\pi_n{=}\texttt{ACK\_CS\_RECV}_x$ — by induction and fairness
- $\pi_n{=}\texttt{ACK_CS_RECV}_x \implies x{\in}$pending_erasable in state s_n
- $x{\in}$pending_erasable in state $s_n \implies \exists p$, $p{>}n$, s.t. $\pi_p{=}\texttt{ERASABLE}_x$
- $\pi_p{=}\texttt{ERASABLE}_x \implies \exists j$, $j{>}p$, s.t. $\pi_j{=}\texttt{ERASE}_x$ — by WF4
- By transitivity, $j{>}i$ and thus the lemma has been proven. □

Lemma A.56 $\quad$ (α is a well-formed and fair execution)
$\wedge$ ($x{\in}$send_cs_recv in state s_i)
$\wedge$ ($\text{timestamp}_x{=}t$, $t{\in}\mathcal{N}$, in state s_i)
$\implies$
$\exists j$, $j{>}i$, s.t $\forall k$, $k{\geq}j$, $t{\notin}$recv_tss in state s_k

Proof:

- $x{\in}$send_cs_recv in state s_i $\implies \exists l$, $l{>}i$, s.t. $\pi_l{=}\texttt{ERASE}_x$ — by Lemma A.55
- $\pi_l{=}\texttt{ERASE}_x \implies$ (($\text{status}_x{=}\text{NONR}$) $\wedge$ ($\text{pending\_ack}_x{=}\text{T}$)) in state s_l
- $\text{timestamp}_x{=}t$, $t{\in}\mathcal{N}$, in state s_i
  $\implies \forall n$, $n{\geq}i$, $\text{timestamp}_x{=}t$ in state s_n — by Lemma A.53
- $\pi_l{=}\texttt{ERASE}_x \implies \exists r$, $r{<}l$, s.t. $\pi_r{=}\texttt{CS\_RECV}_x$ — by Lemma A.34

- π_r=$\texttt{CS\_RECV}_x \implies \nexists q,\ q>r$, s.t. π_q=$\texttt{CS\_RECV}_x$ by Lemma A.42
- (status$_x$=NONR in state s$_l$)
  $\wedge$ ($\nexists q,\ q>l$, s.t. π_q=$\texttt{CS\_RECV}_x$)
  $\wedge$ (pending_ack$_x$=T in state s$_l$)
  $\wedge$ ($\forall n,\ n\geq l$, timestamp$_x$=t in state s$_n$)
  $\implies \exists j,\ j>l$, s.t. π_j=<$\texttt{DLR\_GONE}$,t> by definition of fairness
- π_j=<$\texttt{DLR\_GONE}$,t> $\implies t\notin$recv_tss in state s$_j$
- (timestamp$_x$=t, $t\in\mathcal{N}$, in state s$_j$)
  $\wedge$ ($t\notin$recv_tss in state s$_j$)
  $\wedge$ (π_l=$\texttt{ERASE}_x$)
  $\wedge$ ($l<j$)
  $\implies \forall k,\ k\geq j$, $t\notin$recv_tss in state s$_k$ by Lemma A.54

  and thus the lemma has been proven. □

Lemma A.57 (α is a well-formed execution)
$\wedge$ (π_i=$\texttt{ACK\_ASSIGN}_x$)
$\wedge$ (curr_reqd_dlr=x in state s$_j$)
$\wedge$ ($i\leq j$)
$\implies$
curr_reqd_acked=T in state s$_j$

Proof:

- π_i=$\texttt{ACK\_ASSIGN}_x \implies \exists h,\ h<i$, s.t. π_h=$\texttt{COMMIT}_x$ by Lemma A.38
- π_h=$\texttt{COMMIT}_x \implies$ ((curr_reqd_dlr=x) $\wedge$ (curr_reqd_acked=F)) in state s$_h$
- π_h=$\texttt{COMMIT}_x \implies \nexists k,\ k>h$, s.t. π_k=$\texttt{COMMIT}_x$ by WF1
- (curr_reqd_dlr=x, $x\in\mathcal{N}$, in state s$_h$)
  $\wedge$ (curr_reqd_dlr=x, $x\in\mathcal{N}$, in state s$_j$)
  $\wedge$ ($h<j$)
  $\wedge$ ($\nexists k,\ k>h$, s.t. π_k=$\texttt{COMMIT}_x$)
  $\implies$ ($\nexists l,\ h<l\leq j$, s.t. π_l=$\texttt{COMMIT}_y$ for any y)
  $\wedge$ ($\forall m,\ h\leq m\leq j$, curr_reqd_dlr=$x$ in state s$_m$)
- (π_i=$\texttt{ACK\_ASSIGN}_x$)
  $\wedge$ ($\forall m,\ h\leq m\leq j$, curr_reqd_dlr=$x$ in state s$_m$)
  $\wedge$ ($h<i\leq j$)
  $\implies$ curr_reqd_acked=T in state s$_i$
- (curr_reqd_acked=T in state s$_i$) $\wedge$ ($\nexists l,\ i<l\leq j$, s.t. π_l=$\texttt{COMMIT}_y$ for any y)
  $\implies$ curr_reqd_acked=T in state s$_j$

  and thus the lemma has been proven. □

Lemma A.58 $(\alpha$ is a well-formed execution$)$
$\wedge$ $(\pi_k$=<$\texttt{ASSIGN}_x$,t> for some $t)$
$\Longrightarrow$
status$_x$=UNFL in state s$_k$

Proof:

By contradiction. Assume status$_x \neq$UNFL in state s$_k$.

- π_k=<$\texttt{ASSIGN}_x$,t>
  $\Longrightarrow$ $\nexists h$, $h<k$, s.t. π_h=<$\texttt{ASSIGN}_x$,v> for any v — by Lemma A.30
- Either
  (1) status$_x$=REQD in state s$_k$
  - status$_x$=REQD in state s$_k$ $\Longrightarrow$ $\exists j$, $j \leq k$, s.t. π_j=$\texttt{CS\_REQD}_x$
  - π_j=$\texttt{CS\_REQD}_x$ $\Longrightarrow$ x=send_cs_reqd in state s$_{j-1}$
  - x=send_cs_reqd in state s$_{j-1}$ $\Longrightarrow$ $\exists i$, $i \leq j-1$, s.t. π_i=$\texttt{ACK\_ASSIGN}_x$
  - π_i=$\texttt{ACK\_ASSIGN}_x$ $\Longrightarrow$ $\exists h$, $h<i<k$, s.t. π_h=<$\texttt{ASSIGN}_x$,v> for some v — by Lemma A.29

  This contradiction implies that this case is impossible.

  or

  (2) status$_x$=RECV in state s$_k$
  - status$_x$=RECV in state s$_k$ $\Longrightarrow$ $\exists j$, $j \leq k$, s.t. π_j=$\texttt{CS\_RECV}_x$
  - π_j=$\texttt{CS\_RECV}_x$ $\Longrightarrow$ $x\in$send_cs_recv in state s$_{j-1}$
  - $x\in$send_cs_recv in state s$_{j-1}$ $\Longrightarrow$ $\exists i$, $i \leq j-1$, s.t. π_i=$\texttt{ACK\_ASSIGN}_x$ — by Lemma A.33
  - π_i=$\texttt{ACK\_ASSIGN}_x$ $\Longrightarrow$ $\exists h$, $h<i<k$, s.t. π_h=<$\texttt{ASSIGN}_x$,v> for some v — by Lemma A.29

  But this is a contradiction, so this case must be false.

  or

  (3) status$_x$=NONR in state s$_k$
  - status$_x$=NONR in state s$_k$ $\Longrightarrow$ $\exists j$, $j \leq k$, s.t. π_j=$\texttt{ERASE}_x$
  - π_j=$\texttt{ERASE}_x$ $\Longrightarrow$ $\exists i$, $i<j$, s.t. π_i=$\texttt{ACK\_ASSIGN}_x$ — by Lemma A.36
  - π_i=$\texttt{ACK\_ASSIGN}_x$ $\Longrightarrow$ $\exists h$, $h<i<k$, s.t. π_h=<$\texttt{ASSIGN}_x$,v> for some v — by Lemma A.29

  But this is a contradiction and so also this case must be false.
- All possible cases lead to contradictions, so the original assumption must be false and thus the lemma has been proven. □

Lemma A.59 $(\alpha$ is a well-formed and fair execution$)$
$\wedge$ $(\pi_i$=$\texttt{COMMIT}_x)$
$\Longrightarrow$
$\exists l$, $l>i$, s.t. π_l=$\texttt{ACK\_ASSIGN}_x$

Proof:

- $(\pi_i{=}\texttt{COMMIT}_x) \wedge (\text{current\_ts}{=}t \text{ for some } t{\in}\mathcal{N})$
  $\implies <x,t>\in$pending_ts_assign in state s_i
- $<x,t>\in$pending_ts_assign in state $s_i \implies \exists k,\ k{>}i$, s.t. $\pi_k{=}{<}\texttt{ASSIGN}_x,t{>}$
- $\pi_k{=}{<}\texttt{ASSIGN}_x,t{>} \implies$ pending_ack$_x$=T in state s_k
- $\pi_k{=}{<}\texttt{ASSIGN}_x,t{>} \implies$ status$_x$=UNFL in state s_k — by Lemma A.58
- ((status$_x$=UNFL) $\wedge$ (pending_ack$_x$=T)) in state s_k
  $\implies \exists l,\ l{>}k$, s.t. $\pi_l{=}\texttt{ACK\_ASSIGN}_x$ — by Lemma A.47

  and thus the lemma has been proven. □

Lemma A.60

$$
\begin{array}{ll}
 & (\alpha \text{ is a well-formed execution}) \\
\wedge & (\pi_i{=}\texttt{COMMIT}_x) \\
\wedge & (\text{current\_ts}{=}t,\ t{\in}\mathcal{N}, \text{ in state } s_{i-1}) \\
\implies & \\
 & \forall j,\ j{\geq}i,\ ((<x,t>\in\text{pending\_ts\_assign}) \vee (\text{timestamp}_x{=}t)) \text{ in state } s_j
\end{array}
$$

Proof:

By induction.

- $(\pi_i{=}\texttt{COMMIT}_x) \wedge (\text{current\_ts}{=}t \text{ in state } s_{i-1})$
  $\implies <x,t>\in$pending_ts_assign in state s_i
- $(<x,t>\in\text{pending\_ts\_assign in state } s_{j-1}) \wedge (\pi_j{\neq}{<}\texttt{ASSIGN}_x,t{>})$
  $\implies <x,t>\in$pending_ts_assign in state s_j — by definition of steps(LOT)
- $(<x,t>\in\text{pending\_ts\_assign in state } s_{j-1}) \wedge (\pi_j{=}{<}\texttt{ASSIGN}_x,t{>})$
  $\implies$ timestamp$_x$=t in state s_j — by definition of steps(DLR$_x$)
- $(\text{timestamp}_x{=}t \text{ in state } s_{j-1}) \wedge (\pi_j{\neq}{<}\texttt{ASSIGN}_x,u{>} \text{ for any } u)$
  $\implies$ timestamp$_x$=t in state s_j — by definition of steps(DLR$_x$)
- timestamp$_x$=t in state $s_{j-1} \implies \nexists k,\ k{>}j{-}1$, s.t. $\pi_k{=}{<}\texttt{ASSIGN}_x,u{>}$ for any u — by Lemma A.52
- Therefore,
  $((<x,t>\in\text{pending\_ts\_assign}) \vee (\text{timestamp}_x{=}t))$ in state s_{j-1}
  $\implies ((<x,t>\in\text{pending\_ts\_assign}) \vee (\text{timestamp}_x{=}t))$ in state s_j

  for any possible action π_j in a well-formed execution.
- By induction,
  $\forall j,\ j{\geq}i,\ ((<x,t>\in\text{pending\_ts\_assign}) \vee (\text{timestamp}_x{=}t))$ in state s_j

  and so the lemma has been proven. □

Lemma A.61

$$
\begin{array}{ll}
 & (\alpha \text{ is a well-formed execution}) \\
\wedge & (t{\in}\text{recv\_tss in state } s_i,\ t{\in}\mathcal{N}) \\
\implies & \\
 & \exists x \text{ s.t. } \forall j,\ j{\geq}i,\ (\quad (<x,t>\in\text{pending\_ts\_assign}) \\
 & \qquad\qquad \vee\ (\text{timestamp}_x{=}t)\ ) \quad \text{in state } s_j
\end{array}
$$

Proof:

- $t \in$recv_tss in state $s_i \implies \exists h,\ h<i$, s.t. curr_reqd_ts$=t$ in state s_h
- curr_reqd_ts$=t$, $t \in \mathcal{N}$, in state s_h
  $\implies \exists g,\ g \leq h$, s.t. ($\pi_g$=$\texttt{COMMIT}_x$ for some x) $\wedge$ (current_ts$=t$ in state s_{g-1})
- (π_g=$\texttt{COMMIT}_x$) $\wedge$ (current_ts$=t$ in state s_{g-1})
  $\implies \forall j,\ j \geq g$, ((<$x$,$t$>$\in$pending_ts_assign) $\vee$ (timestamp$_x$$=t$)) in state s_j
  by Lemma A.60
- ($g \leq h < i$)
  $\wedge$ ($\forall j,\ j \geq g$, ((<x,t>$\in$pending_ts_assign) $\vee$ (timestamp$_x$$=t$)) in state s_j)
  $\implies \forall j,\ j \geq i$, ((<$x$,$t$>$\in$pending_ts_assign) $\vee$ (timestamp$_x$$=t$)) in state s_j

  and thus the lemma has been proven. □

Lemma A.62 (α is a well-formed execution)
$\wedge$ ($x \in$send_cs_recv in state s_i)
$\implies$
<x,u>$\notin$pending_ts_assign in state s_i for any u

Proof:

By contradiction. Assume <x,u>$\in$pending_ts_assign in state s_i for some u

- $x \in$send_cs_recv in state $s_i \implies \exists h,\ h \leq i$, s.t. π_h=$\texttt{ACK\_ASSIGN}_x$
  by Lemma A.33
- π_h=$\texttt{ACK\_ASSIGN}_x$
  $\implies \exists g,\ g<h$, s.t. π_g=<$\texttt{ASSIGN}_x$,t> for some t by Lemma A.29
- π_g=<$\texttt{ASSIGN}_x$,t> $\implies$ (<x,t>$\in$pending_ts_assign in state s_{g-1})
  $\wedge$ (<x,t>$\notin$pending_ts_assign in state s_g)
- <x,t>$\in$pending_ts_assign in state s_{g-1}
  $\implies \exists f,\ f \leq g-1$, s.t. ($\pi_f$=$\texttt{COMMIT}_x$) $\wedge$ (current_ts$=t$ in state s_{f-1})
- π_f=$\texttt{COMMIT}_x \implies \nexists j,\ j \neq f$, s.t. π_j=$\texttt{COMMIT}_x$ by WF1
- (<x,t>$\notin$pending_ts_assign in state s_g)
  $\wedge$ ($\nexists j,\ j>f$, s.t. π_j=$\texttt{COMMIT}_x$)
  $\wedge$ ($f<g$)
  $\implies \forall k,\ k \geq g$, <$x$,$t$>$\notin$pending_ts_assign in state s_k
- (<x,u>$\in$pending_ts_assign in state s_i)
  $\wedge$ ($\forall k,\ k \geq g$, <x,t>$\notin$pending_ts_assign in state s_k)
  $\wedge$ ($g<h \leq i$)
  $\implies u \neq t$
- <x,u>$\in$pending_ts_assign in state s_i
  $\implies \exists e,\ e \leq i$, s.t. ($\pi_e$=$\texttt{COMMIT}_x$) $\wedge$ (current_ts$=u$ in state s_{e-1})
- (current_ts$=t$ in state s_{f-1}) $\wedge$ (current_ts$=u$ in state s_{e-1}) $\wedge$ ($u \neq t$)
  $\implies e \neq f$
- $e \neq f \implies \exists e,\ e \neq f$, s.t. π_e=$\texttt{COMMIT}_x$

  But this is a contradiction and so the original assumption must be false. Thus the lemma has been proven. □

Lemma A.63 $(\alpha$ is a well-formed execution$)$
$\wedge$ (curr_reqd_dlr=x, $x \in \mathcal{N}$, in state s_i)
$\wedge$ (($(<x,t> \in$ pending_ts_assign)
$\vee$ (timestamp$_x=t$, $t \in \mathcal{N}$)) in state s_i)
$\Longrightarrow$
curr_reqd_ts=t in state s_i

Proof:

- ((<x,t>$\in$pending_ts_assign) $\vee$ (timestamp$_x=t$, $t \in \mathcal{N}$)) in state s_i
  $\Longrightarrow \exists h$, $h \leq i$, s.t. (π_h=$\texttt{COMMIT}_x$) $\wedge$ (current_ts=t, $t \in \mathcal{N}$, in state s_{h-1})
  by Lemma A.48
- (π_h=$\texttt{COMMIT}_x$) $\wedge$ (current_ts=t in state s_{h-1})
  $\Longrightarrow$ ((curr_reqd_dlr=x) $\wedge$ (curr_reqd_ts=t) $\wedge$ (current_ts=t+1)) in state s_h
- (((curr_reqd_dlr=x, $x \in \mathcal{N}$) $\wedge$ (curr_reqd_ts=t, $t \in \mathcal{N}$)) in state s_{m-1})
  $\wedge$ (current_ts$\neq t$ in state s_{m-1})
  $\wedge$ ($\pi_m \neq \texttt{COMMIT}_x$)
  $\Longrightarrow$ (((curr_reqd_dlr$\neq x$) $\wedge$ (curr_reqd_ts$\neq t$)) in state s_m)
  $\vee$ (((curr_reqd_dlr=x) $\wedge$ (curr_reqd_ts=t)) in state s_m)
- (((curr_reqd_dlr$\neq x$, $x \in \mathcal{N}$) $\wedge$ (curr_reqd_ts$\neq t$, $t \in \mathcal{N}$)) in state s_{m-1})
  $\wedge$ (current_ts$\neq t$, $t \in \mathcal{N}$, in state s_{m-1})
  $\wedge$ ($\pi_m \neq \texttt{COMMIT}_x$)
  $\Longrightarrow$ ((curr_reqd_dlr$\neq x$) $\wedge$ (curr_reqd_ts$\neq t$)) in state s_m
- π_h=$\texttt{COMMIT}_x \Longrightarrow \nexists j$, $j>h$, s.t. π_j=$\texttt{COMMIT}_x$ by WF1
- current_ts=t+1 in state $s_h \Longrightarrow \forall l$, $l \geq h$, current_ts$\geq t$+1 in state s_l
- $\forall l$, $l \geq h$, current_ts$\geq t$+1 in state $s_l \Longrightarrow \forall l$, $l \geq h$, current_ts$\neq t$ in state s_l
- (((curr_reqd_dlr=x, $x \in \mathcal{N}$) $\wedge$ (curr_reqd_ts=t, $t \in \mathcal{N}$)) in state s_h)
  $\wedge$ ($\forall l$, $l \geq h$, current_ts$\neq t$ in state s_l)
  $\wedge$ ($\nexists j$, $j>h$, s.t. π_j=$\texttt{COMMIT}_x$)
  $\Longrightarrow \forall k$, $k \geq h$, (((curr_reqd_dlr$\neq x$) $\wedge$ (curr_reqd_ts$\neq t$)) in state s_k)
  $\vee$ (((curr_reqd_dlr=x) $\wedge$ (curr_reqd_ts=t)) in state s_k)
  by induction
- (curr_reqd_dlr=x in state s_i)
  $\wedge$ ($h \leq i$)
  $\wedge$ ($\forall k$, $k \geq h$, (((curr_reqd_dlr$\neq x$) $\wedge$ (curr_reqd_ts$\neq t$)) in state s_k)
  $\vee$ (((curr_reqd_dlr=x) $\wedge$ (curr_reqd_ts=t)) in state s_k))
  $\Longrightarrow$ curr_reqd_ts=t in state s_i

and thus the lemma has been proven. $\square$

Lemma A.64 $(\alpha$ is a well-formed execution$)$
$\wedge$ (curr_reqd_dlr=x, $x \in \mathcal{N}$, in state s_i)
$\Longrightarrow$
curr_reqd_ts$\neq \perp$ in state s_i

Proof:

- curr_reqd_dlr$=x$, $x\in\mathcal{N}$, in state $s_i \implies \exists j$, $j\leq i$, s.t. π_j=$\texttt{COMMIT}_x$
- (π_j=$\texttt{COMMIT}_x$) $\wedge$ (current_ts$=t$, for some $t\in\mathcal{N}$, in state s_{j-1})
  $\implies \forall k$, $k\geq j$, ((<x,t>$\in$pending_ts_assign) $\vee$ (timestamp$_x$$=t$)) in state s_k
  by Lemma A.60
- ($\forall k$, $k\geq j$, ((<x,t>$\in$pending_ts_assign) $\vee$ (timestamp$_x$$=t$)) in state s_k) $\wedge$ ($j\leq i$)
  $\implies$ ((<x,t>$\in$pending_ts_assign) $\vee$ (timestamp$_x$$=t$)) in state s_i
- (curr_reqd_dlr$=x$ in state s_i)
  $\wedge$ (((<x,t>$\in$pending_ts_assign) $\vee$ (timestamp$_x$$=t$)) in state s_i)
  $\implies$ curr_reqd_ts$=t$ in state s_i by Lemma A.63
- $t\in\mathcal{N} \implies$ curr_reqd_ts$\neq\perp$ in state s_i
  and thus the lemma has been proven. □

Lemma A.65 (α is a well-formed execution)
$\wedge$ ($t\in$recv_tss in state s_i)
$\implies$
$t\in\mathcal{N}$

Proof:

- $t\in$recv_tss in state $s_i \implies$ either

  (1) $\exists h$, $h\leq i$, s.t. (π_h=$\texttt{COMMIT}_x$ for some x)
  $\wedge$ (curr_reqd_ts$\neq\perp$ in state s_{h-1})
  $\wedge$ (curr_reqd_ts$=t$ in state s_{h-1})

  - ((curr_reqd_ts$\neq\perp$) $\wedge$ (curr_reqd_ts$=t$)) in state $s_{h-1} \implies t\in\mathcal{N}$

  or

  (2) $\exists h$, $h\leq i$, s.t. (π_h=$\texttt{ACK\_ASSIGN}_x$)
  $\wedge$ (curr_reqd_dlr$=x$ in state s_{h-1})
  $\wedge$ (curr_reqd_ts$=t$ in state s_{h-1})

  - curr_reqd_dlr$=x$, $x\in\mathcal{N}$, in state s_{h-1}
    $\implies$ curr_reqd_ts$\neq\perp$ in state s_{h-1} by Lemma A.64
  - ((curr_reqd_ts$\neq\perp$) $\wedge$ (curr_reqd_ts$=t$)) in state $s_{h-1} \implies t\in\mathcal{N}$

  or

  (3) $\exists h$, $h\leq i$, s.t. (π_h=<$\texttt{DLR\_GONE}$,u> for some $u\neq t$)
  $\wedge$ (curr_reqd_dlr$\neq\perp$ in state s_{h-1})
  $\wedge$ (curr_reqd_ts$=t$ in state s_{h-1})

  - curr_reqd_dlr$\neq\perp$ in state s_{h-1}
    $\implies$ curr_reqd_dlr$=x$, for some $x\in\mathcal{N}$, in state s_{h-1}
  - curr_reqd_dlr$=x$, for some $x\in\mathcal{N}$, in state s_{h-1}
    $\implies$ curr_reqd_ts$\neq\perp$ in state s_{h-1} by Lemma A.64
  - ((curr_reqd_ts$\neq\perp$) $\wedge$ (curr_reqd_ts$=t$)) in state $s_{h-1} \implies t\in\mathcal{N}$

- Therefore, for all possible cases, the desired result is obtained and thus the lemma has been proven. □

Lemma A.66 $(\alpha$ is a well-formed and fair execution$)$
$\wedge$ $(t \in$ recv_tss in state $s_i)$
$\Longrightarrow$
$\exists j,\ j > i$, s.t. $\forall k,\ k \geq j,\ t \notin$ recv_tss in state s_k

Proof:

- $t \in$ recv_tss $\Longrightarrow t \in \mathcal{N}$ by Lemma A.65
- $(t \in$ recv_tss in state $s_i) \wedge ($recv_tss$=\emptyset$ in state $s_0)$
  $\Longrightarrow \exists h,\ h \leq i$, s.t. $(t \notin$ recv_tss in state $s_{h-1}) \wedge (t \in$ recv_tss in state $s_h)$
- $t \in$ recv_tss in state s_h
  $\Longrightarrow \exists x$ s.t. $\forall l,\ l \geq h$, ($(<x,t> \in$ pending_ts_assign$)$
  $\vee$ $($timestamp$_x = t)$) in state s_l
  by Lemma A.61
- $(t \notin$ recv_tss in state $s_{h-1}) \wedge (t \in$ recv_tss in state $s_h)$
  $\Longrightarrow$ either
  (1) $(\pi_h=\texttt{ACK\_ASSIGN}_y$ for some $y) \wedge ($curr_reqd_ts$=t$ in state $s_{h-1})$
    - $(((<x,t> \in$ pending_ts_assign$) \vee ($timestamp$_x=t))$ in state $s_h)$
      $\wedge$ $(\pi_h=\texttt{ACK\_ASSIGN}_y)$
      $\Longrightarrow$ ($(<x,t> \in$ pending_ts_assign$)$
      $\vee$ $($timestamp$_x=t)$) in state s_{h-1}
    - $(\pi_h=\texttt{ACK\_ASSIGN}_y)$
      $\wedge$ $(t \notin$ recv_tss in state $s_{h-1})$
      $\wedge$ $(t \in$ recv_tss in state $s_h)$
      $\Longrightarrow$ $($curr_reqd_dlr$=y$ in state $s_{h-1})$
      $\wedge$ $(y \in$ send_cs_recv in state $s_h)$
    - $($curr_reqd_ts$=t,\ t \in \mathcal{N}$, in state $s_{h-1})$
      $\wedge$ (($(<x,t> \in$ pending_ts_assign$)$
      $\vee$ $($timestamp$_x=t)$) in state $s_{h-1})$
      $\Longrightarrow$ curr_reqd_dlr$=x$ in state s_{h-1} by Lemma A.50
    - $($curr_reqd_dlr$=y$ in state $s_{h-1}) \wedge ($curr_reqd_dlr$=x$ in state $s_{h-1})$
      $\Longrightarrow x=y$
    - $(y \in$ send_cs_recv in state $s_h) \wedge (x=y)$
      $\Longrightarrow x \in$ send_cs_recv in state s_h

  or

  (2) $(\pi_h=<\texttt{DLR\_GONE},u>$ for some $u \neq t) \wedge ($curr_reqd_ts$=t$ in state $s_{h-1})$
    - $(((<x,t> \in$ pending_ts_assign$) \vee ($timestamp$_x=t))$ in state $s_h)$
      $\wedge$ $(\pi_h=<\texttt{DLR\_GONE},u>$ for some $u \neq t)$
      $\Longrightarrow$ ($(<x,t> \in$ pending_ts_assign$)$
      $\vee$ $($timestamp$_x=t)$) in state s_{h-1}

- $(\text{curr\_reqd\_ts}=t,\ t\in\mathcal{N},\ \text{in state } s_{h-1})$
  $\wedge\ ((\ \ (<x,t>\in\text{pending\_ts\_assign})$
  $\quad \vee\ (\text{timestamp}_x=t)\ )\quad \text{in state } s_{h-1})$
  $\Longrightarrow \text{curr\_reqd\_dlr}=x \text{ in state } s_{h-1}$ by Lemma A.50
- $(\pi_h=<\texttt{DLR\_GONE},u>)$
  $\wedge\ (t\notin\text{recv\_tss in state } s_{h-1})$
  $\wedge\ (t\in\text{recv\_tss in state } s_h)$
  $\wedge\ (\text{curr\_reqd\_dlr}=x \text{ in state } s_{h-1})$
  $\Longrightarrow x\in\text{send\_cs\_recv in state } s_h$

or

(3) $(\pi_h=\texttt{COMMIT}_y \text{ for some } y)\wedge(\text{curr\_reqd\_ts}=t \text{ in state } s_{h-1})$

- $(((<x,t>\in\text{pending\_ts\_assign})\vee(\text{timestamp}_x=t)) \text{ in state } s_h)$
  $\wedge\ (\pi_h=\texttt{COMMIT}_y \text{ for some } y)$
  $\Longrightarrow\ (\ \ (<x,t>\in\text{pending\_ts\_assign})$
  $\quad\vee\ (\text{timestamp}_x=t)\ )\quad \text{in state } s_{h-1}$
- $(\text{curr\_reqd\_ts}=t \text{ in state } s_{h-1})$
  $\wedge\ ((\ \ (<x,t>\in\text{pending\_ts\_assign})$
  $\quad\vee\ (\text{timestamp}_x=t)\ )\quad \text{in state } s_{h-1})$
  $\Longrightarrow \text{curr\_reqd\_dlr}=x \text{ in state } s_{h-1}$ by Lemma A.50
- Either

  (i) $\exists g,\ g\leq h-1, \text{ s.t. } \pi_g=\texttt{ACK\_ASSIGN}_x$
  - $(\pi_g=\texttt{ACK\_ASSIGN}_x)$
    $\wedge\ (\text{curr\_reqd\_dlr}=x \text{ in state } s_{h-1})$
    $\wedge\ (g\leq h-1)$
    $\Longrightarrow \text{curr\_reqd\_acked}=\text{T in state } s_{h-1}$
    by Lemma A.57
  - $(\pi_h=\texttt{COMMIT}_y)$
    $\wedge\ (\text{curr\_reqd\_ts}=t,\ t\in\mathcal{N},\ \text{in state } s_{h-1})$
    $\wedge\ (\text{curr\_reqd\_acked}=\text{T in state } s_{h-1})$
    $\wedge\ (\text{curr\_reqd\_dlr}=x \text{ in state } s_{h-1})$
    $\Longrightarrow x\in\text{send\_cs\_recv in state } s_h$

  or

  (ii) $\nexists g,\ g\leq h-1, \text{ s.t. } \pi_g=\texttt{ACK\_ASSIGN}_x$
  - $\text{curr\_reqd\_dlr}=x,\ x\in\mathcal{N},\ \text{in state } s_{h-1}$
    $\Longrightarrow \exists f,\ f\leq h-1, \text{ s.t. } \pi_f=\texttt{COMMIT}_x$
  - $\pi_f=\texttt{COMMIT}_x$
    $\Longrightarrow \exists e,\ e>f, \text{ s.t. } \pi_e=\texttt{ACK\_ASSIGN}_x$ by Lemma A.59
  - $(\exists e,\ e>f, \text{ s.t. } \pi_e=\texttt{ACK\_ASSIGN}_x)$
    $\wedge\ (\nexists g,\ g\leq h-1, \text{ s.t. } \pi_g=\texttt{ACK\_ASSIGN}_x)$
    $\wedge\ (f\leq h-1)$
    $\wedge\ (\pi_h\neq\texttt{ACK\_ASSIGN}_x)$
    $\Longrightarrow \exists e,\ e>h, \text{ s.t. } \pi_e=\texttt{ACK\_ASSIGN}_x$
  - $\pi_f=\texttt{COMMIT}_x \Longrightarrow \nexists d,\ d>f, \text{ s.t. } \pi_d=\texttt{COMMIT}_x$ by WF1

- $(\pi_h=\texttt{COMMIT}_y) \wedge (\nexists d,\ d>f$, s.t. $\pi_d=\texttt{COMMIT}_x) \wedge (f<h)$
  $\implies x\neq y$
- $\pi_h=\texttt{COMMIT}_y \implies$ curr_reqd_dlr$=y$ in state s_h
- (curr_reqd_dlr$=y$ in state s_h)
  $\wedge$ $(x\neq y)$
  $\wedge$ $(\nexists d,\ d>f$, s.t. $\pi_d=\texttt{COMMIT}_x)$
  $\wedge$ $(f<h)$
  $\implies \forall c,\ c>h$, curr_reqd_dlr$\neq x$ in state s_c
- $(\pi_e=\texttt{ACK\_ASSIGN}_x)$
  $\wedge$ $(e>h)$
  $\wedge$ $(\forall c,\ c>h$, curr_reqd_dlr$\neq x$ in state $s_c)$
  $\implies x\in$send_cs_recv in state s_e

Therefore, for all possible cases, it must be true that
$\exists e,\ e\geq h$, s.t. $x\in$send_cs_recv in state s_e

- $x\in$send_cs_recv in state s_e
  $\implies <x,t>\notin$pending_ts_assign in state s_e — by Lemma A.62
- $(<x,t>\notin$pending_ts_assign in state $s_e)$
  $\wedge$ $(\forall l,\ l\geq h$, $((<x,t>\in$pending_ts_assign$) \vee ($timestamp$_x=t))$ in state $s_l)$
  $\wedge$ $(e\geq h)$
  $\implies$ timestamp$_x=t$ in state s_e
- $((x\in$send_cs_recv in state $s_e) \wedge ($timestamp$_x=t,\ t\in\mathcal{N}))$ in state s_e
  $\implies \exists j,\ j>e$, s.t. $\forall k,\ k\geq j,\ t\notin$recv_tss in state s_k — by Lemma A.56

and thus the lemma has been proven. □

Lemma A.67 (α is a well-formed and fair execution)
$\wedge$ (curr_reqd_dlr$=x$, $x\in\mathcal{N}$, in state s_i)
$\wedge$ (curr_reqd_acked=T in state s_i)
$\wedge$ (recv_tss$\neq\emptyset$ in state s_i)
$\wedge$ $(\nexists j,\ j>i$, s.t. $(\pi_j=\texttt{COMMIT}_y$ for some y$)$
$\wedge$ (curr_reqd_dlr$=x$ in state s_{j-1}))
$\implies$
$\exists k,\ k>i$, s.t. $x\in$send_cs_recv in state s_k

Proof:

By contradiction. Assume $\nexists k,\ k>i$, s.t. $x\in$send_cs_recv in state s_k

- curr_reqd_dlr$=x$, $x\in\mathcal{N}$, in state s_i
  $\implies$ curr_reqd_ts$\neq\perp$ in state s_i — by Lemma A.64
- ((curr_reqd_dlr$=x$, $x\in\mathcal{N}$) $\wedge$ (curr_reqd_acked=T)) in state s_i
  $\implies \exists h,\ h\leq i$, s.t. $\pi_h=\texttt{ACK\_ASSIGN}_x$ — by Lemma A.32
- $\pi_h=\texttt{ACK\_ASSIGN}_x \implies \nexists g,\ g>h$, s.t. $\pi_g=\texttt{ACK\_ASSIGN}_x$ — by Lemma A.41

- (curr_reqd_dlr=x, $x\in\mathcal{N}$, in state s_{m-1})
  $\wedge$ (curr_reqd_ts$\neq\perp$ in state s_{m-1})
  $\wedge$ (curr_reqd_acked=T in state s_{m-1})
  $\wedge$ ($\pi_m\neq\texttt{COMMIT}_y$ for any y)
  $\wedge$ ($\pi_m\neq\texttt{ACK\_ASSIGN}_x$)
  $\wedge$ ($x\notin$send_cs_recv in state s_m)
  $\Longrightarrow$ ($\nexists u$, $u\in\mathcal{N}$, s.t. $(\pi_m=<\texttt{DLR\_GONE},u>)$
  $\wedge$ (recv_tss=$\{u\}$ in state s_{m-1}))
  $\wedge$ (curr_reqd_dlr=x in state s_m)
  $\wedge$ (curr_reqd_ts$\neq\perp$ in state s_m)
  $\wedge$ (curr_reqd_acked=T in state s_m)

- (curr_reqd_dlr=x, $x\in\mathcal{N}$, in state s_i)
  $\wedge$ (curr_reqd_ts$\neq\perp$ in state s_i)
  $\wedge$ (curr_reqd_acked=T in state s_i)
  $\wedge$ ($\nexists j$, $j>i$, s.t. ($\pi_j=\texttt{COMMIT}_y$ for some y)
  $\wedge$ (curr_reqd_dlr=x in state s_{j-1}))
  $\wedge$ ($\nexists g$, $g>i$, s.t. $\pi_g=\texttt{ACK\_ASSIGN}_x$)
  $\wedge$ ($\nexists k$, $k>i$, s.t. $x\in$send_cs_recv in state s_k)
  $\Longrightarrow$ ($\forall c$, $c\geq i$, curr_reqd_dlr=x in state s_c)
  $\wedge$ ($\nexists d$, $d>i$, s.t. $(\pi_d=<\texttt{DLR\_GONE},u>)$ $\wedge$
  $\wedge$ (recv_tss=$\{u\}$ in state s_{d-1}) for any u)
  by induction

- (curr_reqd_dlr=x, $x\in\mathcal{N}$, in state s_{m-1})
  $\wedge$ (recv_tss=A in state s_{m-1})
  $\wedge$ ($\pi_m\neq\texttt{COMMIT}_y$ for any y)
  $\wedge$ ($\pi_m\neq\texttt{ACK\_ASSIGN}_x$)
  $\wedge$ ($\nexists u$, $u\in\mathcal{N}$, s.t. $(\pi_m=<\texttt{DLR\_GONE},u>)$ $\wedge$ (recv_tss=$\{u\}$ in state s_{m-1}))
  $\Longrightarrow$ recv_tss$\subseteq A$ in state s_m by definition of steps(LOT)

- (recv_tss=R in state s_i)
  $\wedge$ ($R\neq\emptyset$)
  $\wedge$ ($\forall c$, $c\geq i$, curr_reqd_dlr=x in state s_c)
  $\wedge$ ($\nexists j$, $j>i$, s.t. ($\pi_j=\texttt{COMMIT}_y$ for some y)
  $\wedge$ (curr_reqd_dlr=x in state s_{j-1}))
  $\wedge$ ($\nexists g$, $g>i$, s.t. $\pi_g=\texttt{ACK\_ASSIGN}_x$)
  $\wedge$ ($\nexists d$, $d>i$, s.t. $(\pi_d=<\texttt{DLR\_GONE},u>)$
  $\wedge$ (recv_tss=$\{u\}$ in state s_{d-1}) for any u)
  $\Longrightarrow$ $\forall b$, $b\geq i$, recv_tss$\subseteq R$ in state s_b by induction

- ($\forall b$, $b\geq i$, recv_tss$\subseteq R$ in state s_b)
  $\wedge$ ($R\neq\emptyset$)
  $\wedge$ ($\nexists d$, $d>i$, s.t. $(\pi_d=<\texttt{DLR\_GONE},u>)$
  $\wedge$ (recv_tss=$\{u\}$ in state s_{d-1}) for any u)
  $\Longrightarrow$ $\exists v$, $v\in R$, s.t. $\forall a$, $a\geq i$, $v\in$recv_tss in state s_a

- $v \in R$ in state s_i
  $\Longrightarrow \exists n,\ n>i$, s.t. $\forall q,\ q \geq n,\ v \notin$recv_tss in state s_q by Lemma A.66

  But this contradicts the earlier deduction that
  $\forall a,\ a \geq i,\ v \in$recv_tss in state s_a
  and so the original assumption must be false and the lemma has been proven.
  □

Lemma A.68

$$
\begin{array}{ll}
 & (\alpha \text{ is a well-formed and fair execution}) \\
\wedge & (\text{curr\_reqd\_acked=T in state } s_l) \\
\wedge & (\text{curr\_reqd\_dlr}=x,\ x \in \mathcal{N}, \text{ in state } s_l) \\
\wedge & (\text{recv\_tss} \neq \emptyset \text{ in state } s_l) \\
\Longrightarrow & \\
 & \exists r,\ r>l, \text{ s.t. } x \in \text{send\_cs\_recv in state } s_r
\end{array}
$$

Proof:

- Either

  (1) $\exists r,\ r>l$, s.t. $(\pi_r=\texttt{COMMIT}_y$ for some $y)$
  $\wedge$ (curr_reqd_dlr$=x$ in state s_{r-1})

  - curr_reqd_dlr$=x$, $x \in \mathcal{N}$, in state s_{r-1}
    $\Longrightarrow$ curr_reqd_ts$\neq\perp$ in state s_{r-1} by Lemma A.64
  - (curr_reqd_dlr$=x$ in state s_l)
    $\wedge$ (curr_reqd_dlr$=x$ in state s_{r-1})
    $\wedge$ $(r>l)$
    $\Longrightarrow \nexists q,\ l \leq q \leq r-1$, s.t. $\pi_q=\texttt{COMMIT}_z$ for any z
  - (curr_reqd_acked=T in state s_l)
    $\wedge$ $(\nexists q,\ l \leq q \leq r-1$, s.t. $\pi_q=\texttt{COMMIT}_z$ for any $z)$
    $\Longrightarrow$ curr_reqd_acked=T in state s_{r-1}
  - $(\pi_r=\texttt{COMMIT}_y$ for some $y)$
    $\wedge$ (curr_reqd_dlr$=x$ in state s_{r-1})
    $\wedge$ (curr_reqd_ts$\neq\perp$ in state s_{r-1})
    $\wedge$ (curr_reqd_acked=T in state s_{r-1})
    $\Longrightarrow x \in$send_cs_recv in state s_r

  or

  (2) $\nexists m,\ m>l$, s.t. $(\pi_m=\texttt{COMMIT}_y$ for some $y)$
  $\wedge$ (curr_reqd_dlr$=x$ in state s_{m-1})

  - (curr_reqd_dlr$=x$, $x \in \mathcal{N}$, in state s_l)
    $\wedge$ (curr_reqd_acked=T in state s_l)
    $\wedge$ (recv_tss$\neq\emptyset$ in state s_l)
    $\wedge$ $(\nexists m,\ m>l$, s.t. $(\pi_m=\texttt{COMMIT}_y$ for some y)
    $\wedge$ (curr_reqd_dlr$=x$ in state s_{m-1}))
    $\Longrightarrow \exists r,\ r>l$, s.t. $x \in$send_cs_recv in state s_r by Lemma A.67
- The desired result follows for both possible cases, and so the lemma has been proven. □

Theorem A.69 $\quad (\alpha$ is a well-formed and fair execution$)$
$\wedge \quad (\pi_i = \texttt{COMMIT}_x)$
$\Longrightarrow$
$\exists j,\ j > i$, s.t. $\pi_j = \texttt{ERASE}_x$

Proof:

- $\pi_i = \texttt{COMMIT}_x \Longrightarrow \exists l,\ l > i$, s.t. $\pi_l = \texttt{ACK\_ASSIGN}_x$ by Lemma A.59
- $\pi_i = \texttt{COMMIT}_x \Longrightarrow \nexists k,\ k > i$, s.t. $\pi_k = \texttt{COMMIT}_x$ by WF1
- Either
  (1) $\exists m,\ i < m < l$, s.t. $\pi_m = \texttt{COMMIT}_y$ for some $y \neq x$
  - $(\pi_m = \texttt{COMMIT}_y$ for some $y \neq x) \wedge (\nexists k,\ k > i$, s.t. $\pi_k = \texttt{COMMIT}_x)$
    $\Longrightarrow$ curr_reqd_dlr$\neq x$ in state s_{l-1}
  - (curr_reqd_dlr$\neq x$ in state s_{l-1}) $\wedge$ $(\pi_l = \texttt{ACK\_ASSIGN}_x)$
    $\Longrightarrow$ $x \in$send_cs_recv in state s_l

  or

  (2) $\nexists m,\ i < m < l$, s.t. $\pi_m = \texttt{COMMIT}_y$ for any y
  - $\pi_i = \texttt{COMMIT}_x$
    $\Longrightarrow$ ((curr_reqd_dlr$=x$) $\wedge$ (curr_reqd_acked=F)) in state s_i
  - $\pi_l = \texttt{ACK\_ASSIGN}_x$
    $\Longrightarrow$ $\nexists h,\ h < l$, s.t. $\pi_h = \texttt{ACK\_ASSIGN}_x$ by Lemma A.41
  - (((curr_reqd_dlr$=x$) $\wedge$ (curr_reqd_acked=F)) in state s_{n-1})
    $\wedge$ $(\pi_n \neq \texttt{COMMIT}_y$ for any $y)$
    $\wedge$ $(\pi_n \neq \texttt{ACK\_ASSIGN}_x)$
    $\Longrightarrow$ ((curr_reqd_dlr$=x$) $\wedge$ (curr_reqd_acked=F)) in state s_n
  - (((curr_reqd_dlr$=x$) $\wedge$ (curr_reqd_acked=F)) in state s_i)
    $\wedge$ $(\nexists m,\ i < m < l$, s.t. $\pi_m = \texttt{COMMIT}_y$ for any $y)$
    $\wedge$ $(\nexists h,\ h < l$, s.t. $\pi_h = \texttt{ACK\_ASSIGN}_x)$
    $\Longrightarrow$ ((curr_reqd_dlr$=x$) $\wedge$ (curr_reqd_acked=F)) in state s_{l-1}
    by induction
  - Either
    (i) recv_tss$=\emptyset$ in state s_{l-1}
    - (curr_reqd_dlr$=x$ in state s_{l-1})
      $\wedge$ (recv_tss$=\emptyset$ in state s_{l-1})
      $\wedge$ $(\pi_l = \texttt{ACK\_ASSIGN}_x)$
      $\Longrightarrow$ $x \in$send_cs_recv in state s_l

    or

    (ii) recv_tss$\neq\emptyset$ in state s_{l-1}
    - (curr_reqd_dlr$=x$ in state s_{l-1})
      $\wedge$ (recv_tss$\neq\emptyset$ in state s_{l-1})
      $\wedge$ $(\pi_l = \texttt{ACK\_ASSIGN}_x)$
      $\Longrightarrow$ (curr_reqd_acked=T in state s_l)
      $\wedge$ (curr_reqd_dlr$=x$ in state s_l)
      $\wedge$ (recv_tss$\neq\emptyset$ in state s_l)

- $\quad$ (curr_reqd_acked=T in state s_l)
  $\wedge$ (curr_reqd_dlr=x in state s_l)
  $\wedge$ (recv_tss$\neq\emptyset$ in state s_l)
  $\implies \exists r,\ r>l$, s.t. $x\in$send_cs_recv in state s_r

  by Lemma A.68

- Therefore, for all possible cases, it is true that
  $\exists r,\ r\geq l$, s.t. $x\in$send_cs_recv in state s_r
- $x\in$send_cs_recv in state s_r
  $\implies \exists j,\ j>r$, s.t. $\pi_j=\texttt{ERASE}_x$ by Lemma A.55

  and thus the theorem has been proven. $\square$

Bibliography

[1] Rakesh Agrawal. A Parallel Logging Algorithm for Multiprocessor Database Machines. In *Proc 4th Int'l Workshop on Database Machines*, pages 256–276, Grand Bahama Island, March 1985.

[2] Rakesh Agrawal and David J. DeWitt. Recovery Architectures for Multiprocessor Database Machines. In *Proc SIGMOD '85 Conf*, pages 131–145, Austin, Texas, May 1985.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[4] Joel F. Bartlett. A NonStop Kernel. In *Proc 8th Symp. on Operating System Principles*, pages 22–29, December 1981.

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[6] Ian F. Blake. *An Introduction to Applied Probability*. John Wiley & Sons, New York, New York, 1979.

[7] Haran Boral. Design Considerations for 1990 Database Machines. In *Proc IEEE Compcon '86 Conf*, pages 370–373, San Francisco, California, March 1986.

[8] Haran Boran, David J. DeWitt, Dina Friedland, Nancy F. Jarrell, and W. Kevin Wilkinson. Implementation of the Database Machine DIRECT. *IEEE Trans on Software Engineering*, SE-8(6):533–543, November 1982.

[9] George Copeland, Tom Keller, Ravi Krishnamurthy, and Marc Smith. The Case for Safe RAM. In *Proc 15th Int'l Conf on Very Large Databases*, pages 327–335, Amsterdam, The Netherlands, August 1989.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[11] William Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Lethin, Peter Nuth, Scott Wills, Paul Carrick, and Greg Fyler. The J-Machine: A Fine-Grain Concurrent Computer. In *Proc IFIP 11th World Computer Congress*, pages 1147–1153, San Francisco, California, August 1989.

[12] William J. Dally, J.A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[13] Dean S. Daniels, Alfred Z. Spector, and Dean S. Thompson. Distributed Logging for Transaction Processing. In *Proc SIGMOD '87 Conf*, pages 82–96, San Francisco, California, May 1987.

[14] David J. DeWitt. DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems. *IEEE Trans on Computers*, C-28(6):395–406, June 1979.

[15] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation Techniques for Main Memory Database Systems. In *Proc SIGMOD '84 Conf*, pages 1–8, Boston, Massachusetts, June 1984.

[16] Susanne Englert, Jim Gray, Terrye Kocher, and Praful Shah. A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases. Technical Report 89.4, Tandem Computers Inc., May 1989.

[17] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Comm. of the ACM*, 19(11):624–633, November 1976.

[18] Anon. et al. A Measure of Transaction Processing Power. *Datamation*, pages 112–118, April 1985.

[19] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans on Computers*, C-32(2):175–189, February 1983.

[20] Jim Gray. Notes on Data Base Operating Systems. Set of course notes, 1977.

[21] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, San Mateo, California, 1991.

[22] Jim Gray, Bob Good, Dieter Gawlick, Pete Homan, and Harald Sammer. One Thousand Transactions per Second. In *Proc IEEE Compcon '85 Conf*, pages 96–101, San Francisco, California, February 1985.

[23] Jim Gray, Bob Horst, and Mark Walker. Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput. In *Proc 16th Int'l Conf on Very Large Databases*, pages 148–161, Brisbane, Australia, August 1990.

[24] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[25] Jim Gray and Frank Putzolu. The 5 Minute Rule for Trading Memory for Disc Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *Proc SIGMOD '87 Conf*, pages 395–398, San Francisco, California, May 1987.

[26] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[27] The Tandem Database Group. NonStop SQL: A Distributed, High-Performance High-Availability Implementation of SQL. Technical Report 87.4, Tandem Computers Inc., April 1987.

[28] The Tandem Performance Group. A Benchmark of NonStop SQL on the Debit Credit Transaction. In *Proc SIGMOD '88 Conf*, pages 337–341, Chicago, Illinois, June 1988.

[29] Linley Gwennap. 1992 in Review: The Top RISC Processors. *Microprocessor Report*, 6(17), 1992.

[30] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[31] Robert B. Hagmann. A Crash Recovery Scheme for a Memory-Resident Database System. *IEEE Trans on Computers*, C-35(9):839–843, September 1986.

[32] Robert B. Hagmann and Hector Garcia-Molina. Implementing Long Lived Transactions Using Log Record Forwarding. Technical Report CSL-91-2, Xerox Palo Alto Research Center, February 1991.

[33] Robert Holbrook. NonStop SQL - A Distributed Relational DBMS for OLTP. In *Proc IEEE Compcon '88 Conf*, pages 418–421, San Francisco, California, February 1988.

[34] John Kaunitz and Louis Van Ekert. Audit Trail Compaction for Database Recovery. *Comm. of the ACM*, 27(7):678–683, July 1984.

[35] John S. Keen and William J. Dally. Performance Evaluation of Ephemeral Logging. In *Proc 1993 ACM SIGMOD Int'l Conf on Management of Data*, pages 187–196, Washington, D.C., May 1993.

[36] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[37] Walter H. Kohler. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. *ACM Computing Surveys*, 13(2):149–183, June 1981.

[38] Akhil Kumar. A Crash Recovery Algorithm Based on Multiple Logs that Exploits Parallelism. In *Proc 2nd IEEE Symp on Parallel and Distributed Processing*, pages 156–159, Dallas, Texas, December 1990.

[39] Tobin J. Lehman and Michael J. Carey. A Recovery Algorithm for A High-Performance Memory-Resident Database System. In *Proc SIGMOD '87 Conf*, pages 104–117, San Francisco, California, May 1987.

[40] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetime of Objects. *Comm. of the ACM*, 26(6):419–429, June 1983.

[41] David B. Lomet. Recovery for Shared Disk Systems Using Multiple Redo Logs. Technical Report CRL 90/4, DEC, October 1990.

[42] Nancy Lynch. I/O Automata: A Model for Discrete Event Systems. Technical Report MIT/LCS/TM-351, MIT, March 1988.

[43] Nancy A. Lynch and Mark R. Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Also, appeared as Technical Memo MIT/LCS/TM-373.

[44] Jeff Moad. Relational Takes on OLTP. *Datamation*, pages 36–39, May 1991.

[45] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwartz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans on Database Systems*, 17(1):94–162, March 1992.

[46] Philip A. Neches. The Anatomy of a Data Base Computer System. In *Proc IEEE Compcon '85 Conf*, pages 252–254, San Francisco, California, February 1985.

[47] Philip A. Neches. The Anatomy of a Data Base Computer System - Revisited. In *Proc IEEE Compcon '86 Conf*, pages 374–377, San Francisco, California, March 1986.

[48] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224–235, San Diego, California, May 1993. IEEE.

[49] S.C. North and J.H. Reppy. Concurrent Garbage Collection on Stock Hardware. In *Proc Functional Programming Languages and Computer Architecture (Springer-Verlag LNCS 274)*, pages 113–133, Portland, Oregon, September 1987.

[50] Peter R. Nuth and William J. Dally. The J-Machine Network. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 420–423. IEEE, October 1992.

[51] David Patterson, Peter Chen, Garth Gibson, and Randy Katz. Introduction to Redundant Arrays of Inexpensive Disks (RAID). In *Proc IEEE Compcon '89 Conf*, pages 112–117, San Francisco, California, February 1989.

[52] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc SIGMOD '88 Conf*, pages 109–116, Chicago, Illinois, June 1988.

[53] Don Pierce, Marty Czekalski, and Charles Cassidy. Solid State Technology Answers The Need For High Performance I/O. *Computer Technology Review*, pages 40–45, Summer 1993.

[54] David Reed and Liba Svobodova. SWALLOW: A Distributed Data Storage System for a Local Network. In *Local Networks for Computer Communications: Proceedings of the IFIP Working Group 6.4 International Workshop on Local Networks*, pages 355–373, Zurich, Switzerland, August 1980. North-Holland.

[55] Mike Ricciuti. Teradata Adds Power for OLTP. *Datamation*, pages 89–90, May 1991.

[56] Mendel Rosenblum and John K. Ousterhout. The LFS Storage Manager. In *Proc Summer '90 USENIX Technical Conf*, Anaheim, California, June 1990.

[57] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proc of the 13th ACM Symp on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, October 1991.

[58] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans on Computer Systems*, 10(1):26–52, February 1992.

[59] Ravi Sharma and Mary Lou Soffa. Parallel Generational Garbage Collection. In *Proc Object-Oriented Programming Systems, Languages, and Applications Conf*, pages 16–32, Phoenix, Arizona, October 1991.

[60] Jack Shemer and Phil Neches. The Genesis of a Database Computer. *IEEE Computer Magazine*, 17(11):42–56, November 1984.

[61] Emy Tseng and David Reiner. Parallel Database Processing on the KSR1 Computer. In *Proc 1993 ACM SIGMOD Int'l Conf on Management of Data*, pages 453–455, Washington, D.C., May 1993.

[62] David Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proc ACM SIGSOFT/SIGPLAN Software Engineering Symp on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, April 1984.

[63] Benjamin Zorn. Comparing Mark-and-sweep and Stop-and-copy Garbage Collection. In *Proc ACM Symp on Lisp and Functional Programming*, pages 87–98, Nice, France, June 1990.