StarBase v2.2 Implementation Details

Matthew Lehr
mrl6a@uvacs.virginia.edu

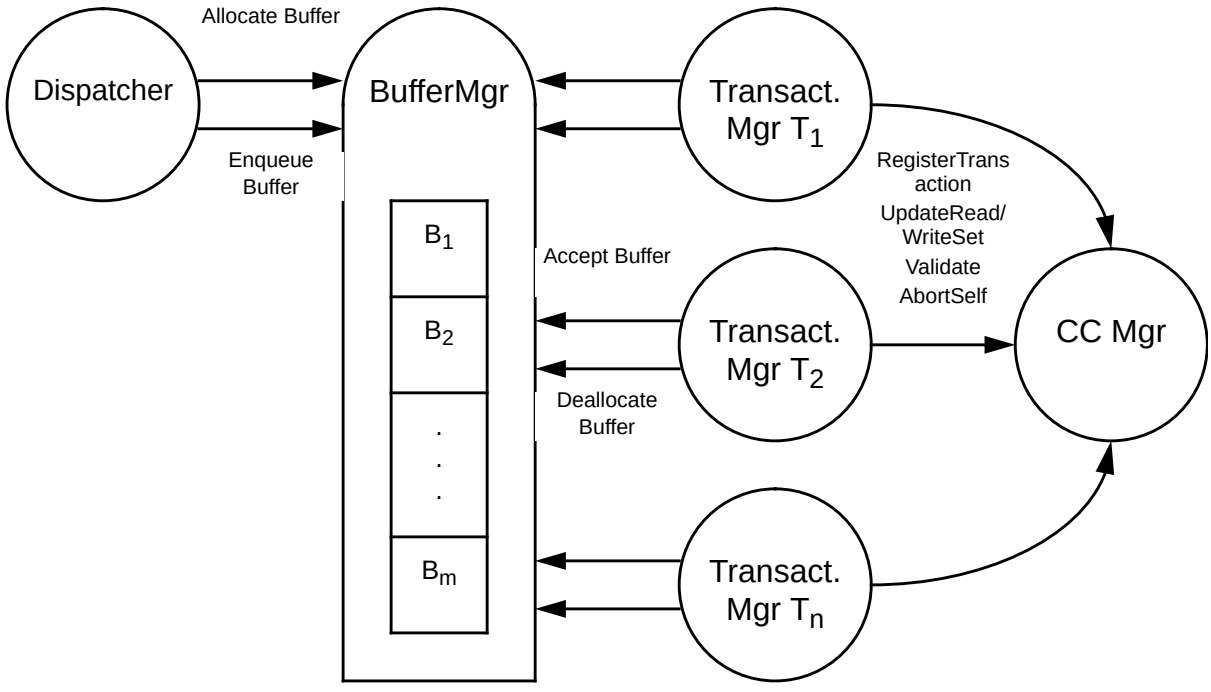
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
July 1993

I. Introduction

The changes required for new front end for the StarBase database are quite extensive, encompassing 4500 new lines of code as well as a substantial amount of modifications to existing code.

The StarBase server itself underwent a major overhaul and now consists of four major areas: the Dispatcher, the Buffer Manager, the Transaction Managers, and the Concurrency Control Manager. Note that the Dispatcher corresponds to the Server Root Thread and the Transaction Managers correspond to the Workers in the previous implementation of StarBase.

StarBase Architecture Overview



In a nutshell, StarBase works as follows: the Dispatcher receives transaction requests directly from StarBase clients, and forwards those requests to the Buffer Manager. The Buffer Manager then determines the service order of the requests according to a *priority metric* (discussed below). When they become available to accept new transactions, the Transaction Managers obtain the next highest priority transaction from the Buffer Manager, which they service either to completion or until the deadline is up.

The behavior of each area is detailed below.

II. The Dispatcher

The Dispatcher (formerly the Server Root Thread) is somewhat of a misnomer. In the original implementation, the Server Root Thread actively assigned (or *dispatched*) incoming transactions to Workers. In the new implementation, however, there is no *a priori* assignment of transactions to Transaction Managers. Instead, the assignment has been broken up into two parts: first, the Buffer Manager determines the service order according to the priority metric, and subsequently, when it becomes ready, a Transaction Manager accepts the highest priority unserviced transaction.

The Dispatcher itself performs some minimal deadline enforcement. When a transaction arrives, the Dispatcher adds a correction to the transaction's deadline. Ideally, the correction should be an estimator of how long it would take StarBase to send a single deadline abort message back to the requesting client. (Currently, StarBase employs a constant -10 ms correction.) If, after adding the correction, the Dispatcher determines that the transaction cannot be serviced in time, the transaction is rejected out of hand. Transactions which can be serviced are handed directly to the Buffer Manager.

III. The Buffer Manager

The Buffer Manager's primary function is to allow the Dispatcher and Transaction Managers to access the buffers containing transactions in a mutually exclusive manner. While the current implementation of the Buffer Manager is a message-based server, the Buffer Manager could also be implemented using synchronization primitives (NOTE: although [Toku91] discusses RT-Mach synchronization mechanisms, there are none for the RT-Mach MK78 kernel [Loep92]).

There are four services which the Buffer Manager provides:

Allocate provides an empty buffer into which an incoming transaction may be received.
Enqueue inserts a buffer containing a new transaction into the service queue.
Accept provides a buffer containing the highest priority unserviced transaction.
Deallocate returns a buffer that has been accepted to the free pool.

The Dispatcher uses only the allocate and enqueue services and the Transaction Managers use only the accept and deallocate services. For efficiency, the enqueue/allocate and deallocate/accept service pairs have been combined. The Dispatcher will always allocate an empty buffer after enqueuing a new

transaction; a Transaction Manager will always accept a new transaction after deallocating the transaction it has completed.

In order to minimize the amount of copying, the unit of transfer between the Buffer Manager and its clients is a reference to a buffer rather than the buffer itself. This allows the Dispatcher to copy an incoming transaction into its *buffer of final destination*: the Transaction Manager will service the transaction out of the same buffer into which it was received.

In order to minimize the response time for transactions which cannot be processed in time, the allocate service is designed so that it never blocks. Under heavy load, the Dispatcher must always have a free buffer into which it can receive (and possibly reject) transactions. To achieve this end, the Buffer Manager has the ability to *force out* the lowest priority unserviced transaction in the service queue even though its deadline has technically not expired. The Buffer Manager replies to its client in the expectation that the Server will not be able to process the transaction in time. Note that the Buffer Manager cannot force out transactions which Transaction Managers are actively processing. (Deadlines of running transactions are enforced by separate Deadline Managers (see below).)

Another important function of the Buffer Manager is to determine the service order of incoming transactions. The Buffer Manager maintains a priority service queue into which new transactions are inserted according to a *priority metric*. Transactions are removed from the queue either when a Transaction Manager accepts the next highest priority transaction or when the Buffer Manager must perform a force out on the lowest priority transaction.

The priority metric, in the context of StarBase, is a function which maps priority and deadline pairs onto nonnegative integers. The problem of determining the service order for *n* transactions can then be reduced to forming a total ordering of transactions based on their priority metric values. The choice of priority metric may be different to achieve different goals, but the priority metrics employed by each area in the database should be identical (in particular, the Concurrency Control Manager employs the same priority metric as the Buffer Manager). For example, to minimize the total deadlines missed, one might choose the priority metric *f (priority, deadline) = deadline* and form a total ordering with the minimum *f()* as the highest priority. Currently, StarBase employs a priority metric which orders transactions by priority first, then by deadline within the same priority level. Note that while this implementation is simple and requires minimal computation (it only uses compares and does not strictly compute a non-negative integer), a priority metric which gives more weight to the deadline should be found.

The final function of the Buffer Manager is to assign a unique transaction id (*tid*) to each transaction that is accepted by a Transaction Manager. The tid is generally only used for debugging, although the Concurrency Control Manager has other purposes for it (see below).

IV. The Transaction Manager

The Transaction Managers, which correspond to the Workers of the previous implementation, are responsible for initiating the active processing of transactions. The Transaction Manager consists of two parts: the code which actually performs the transaction processing and the code which enforces the transaction deadline. The bulk of the transaction processing is performed by code which is taken from MRDB, but includes David George's indexing code [Geor93], and some initialization.

Much like the former implementation of StarBase, Transaction Managers may run at different (RT-Mach) priority levels. However, instead of fixing its RT-Mach priority level at initialization time, a Transaction Manager adjusts its priority according to the StarBase priority of each transaction it accepts.

The Transaction Manager determines the appropriate MRDB function to call based on the transaction operation type. The MRDB functions must behave in accordance with several rules in order to work correctly:

1) If the MRDB function references the contents of a relation, it must abide by the Concurrency Manager protocol (see below).
2) The MRDB function must return control to the Transaction Manager only under the following conditions: a) the MRDB function completes successfully, b) the MRDB function encounters an error, c) the MRDB function is aborted due to a concurrency control conflict, or d) the transaction deadline expires.
3) If the MRDB function completes successfully, it must reply to the client and return OK to the Transaction Manager.
4) If the MRDB function encounters an error, it must reply to the client with the appropriate error code and return the error code to the Transaction Manager.
5) If the MRDB function is aborted due to a concurrency control conflict, it must *not* reply to the client and must return CONFLICT_ABORT to the Transaction Manager.
6) If the deadline expires, the MRDB function must *not* reply to the client and must return DEADLINE_ABORT to the Transaction Manager.

Before any transaction processing begins, the Transaction Manager starts a Deadline Handler. The Deadline Handler's purpose is to abort the transaction and reply to the client if and when the deadline expires. The deadline expiration is detected by arming and waiting on a RT-Mach timer. When the timer expires, the Deadline Manager wakes up and replies directly to the transaction's client in order to notify the client of the abort as soon as possible. Note that if the correction applied to the transaction's deadline by the Dispatcher (see above) is accurate, the client should receive the abort reply from the Deadline Manager in advance of the deadline.

Although deadline expiration seems to warrant a fully asynchronous abort of the Transaction Manager by the Deadline Handler, the MRDB code is not designed to support asynchronous aborts. Unfortunately, this entails that the Transaction Manager go through some contortions to ensure correct handling of expired deadlines. In particular, the Deadline Handler, instead of summarily terminating the Transaction Manager, simply marks it as aborted. The Transaction Manager, in turn, must periodically poll its status to determine whether the deadline is up. Note that even though the client is notified immediately by the Deadline Handler when the deadline expires, the Transaction Manager may continue processing until the next poll point before the transaction actually aborts.

V. The Concurrency Control Manager

The Concurrency Control Manager was rewritten in its entirety not only to provide a more efficient implementation and to fix logical errors in the code, but also to provide code which corresponds to the actual concepts involved in an optimistic concurrency control mechanism. Obtusely (and inappropriately)

named data structures from the old implementation such as LOCK_INFO, SET_LOCK, and LMDATA have been replaced by data structures which correspond directly to information required by an optimistic concurrency control scheme (e.g. DataSets (including both readSet and writeSet fields), ConflictSet, etc.).

The scheme implemented in StarBase is the optimistic concurrency control algorithm OPT-WAIT-50 proposed by Haritsa [144]. The operations provided by the original implementation, check_in, lock, and validate, have been replaced by the operations RegisterTransaction, RegisterReferenceIntent, UpdateReadSet, UpdateWriteSet, Validate, ScheduleAbort, and AbortSelf. The operations are detailed below:

RegisterTransaction must be called by each MRDB function before it references the contents of any relation. RegisterTransaction provides the Concurrency Control Manager with information about the caller (deadline, priority, and thread) and about the initial relation to be referenced. The Concurrency Control Manager requires the transaction deadline and priority in order to determine the CHP and CLP sets (see below), and it requires the thread in order to enforce mutual exclusion during the validation process.

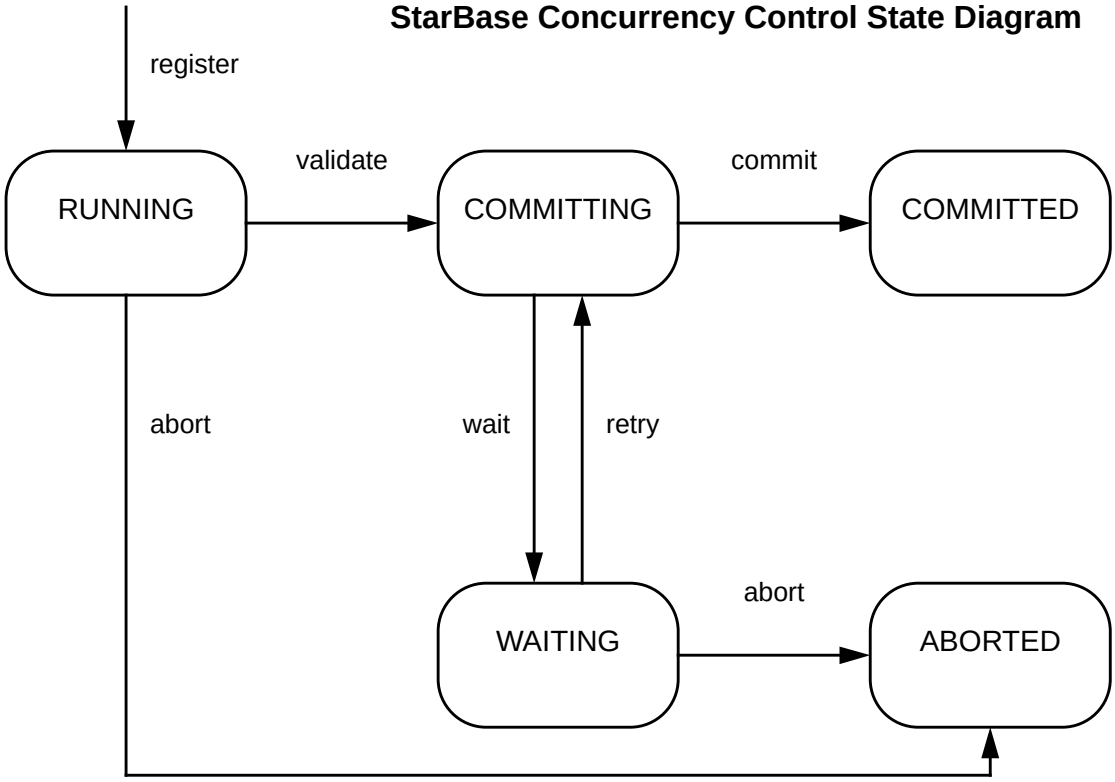
RegisterReferenceIntent is used to notify the Concurrency Control Manager that the transaction intends to reference other relations beyond the one initially specified by RegisterTransaction. As is the case with the initial relation specified with RegisterTransaction, the transaction is required to specify whether it intends to read and/or write to the relation in question. A particular transaction should use RegisterReferenceIntent at most once for each relation.

UpdateReadSet and **UpdateWriteSet** are used to notify the Concurrency Control Manager that the transaction intends to read or write a particular tuple. It is particularly important that transactions update the data set prior to actually performing the read or write. This ensures the correct calculation of conflict sets for transactions which may be validating while the current transaction is reading or writing. As in the previous implementation, these operations are kept as light weight as possible since they are called frequently. The Update operations do not call directly into the Concurrency Control Manager, rather they modify the dataSet data structures directly.

Validate is used to validate the transaction via the Concurrency Control Manager. Validation entails computing the conflict set for the transaction and then deciding whether to wait or to commit. If the transaction commits, the Concurrency Control Manager must abort all transactions with which it conflicts.

ScheduleAbort is intended to be used by the Concurrency Control Manager to abort conflicting transactions during validation. The operation was provided externally so that a Deadline Manager can notify the Concurrency Control Manager when a transaction has deadline aborted. For reasons of modularity, it was decided not to use ScheduleAbort in this manner, but the implementation has been left in as it would be particularly useful for a fully asynchronous abort. Note that Schedule Abort does not abort a transaction in its entirety since UpdateReadSet and UpdateWriteSet modify Concurrency Control data structures directly. Transactions with aborts scheduled are unlinked from the Concurrency Control internal data structures, but are not freed until the aborted transaction actually calls AbortSelf.

AbortSelf is provided so that a transaction may unregister itself from the Concurrency Control Manager without validating. Currently AbortSelf is used when the transaction detects that an abort has been scheduled.



Since the bulk of the concurrency control work in OPT-WAIT-50 must be performed at the validation point, the Concurrency Control Manager must be able to efficiently locate potentially conflicting transactions, calculate conflict sets, suspend waiting transactions, and abort conflicting transactions. Three data structures are required: a transaction hash table which holds transaction information, a relation table to hold the data sets associated with each relation, and a queue of transactions waiting to validate.

The hash table and wait queue contain transaction information. Each transaction data structure contains vital information about the transaction, including tid (see Buffer Manager, above), deadline, priority, thread, and status, as well as the transaction's conflict set and data sets. Extra links are provided in the data structure so that the transaction may be linked into the transaction hash table and wait queue without the overhead of malloc'ing or copying.

The relation table contains all of the data sets which refer to each relation. Each data set contains a read set and write set. If the transaction has specified (via RegisterTransaction or RegisterReferenceIntent) that it will only read or only write to the relation, then the write set or read set, respectively, of that transaction's data set will be empty. The read and write sets are simply bitmaps with all bits initially zero. As the transaction calls UpdateReadSet or UpdateWriteSet to record the read or write of tuple *i* in the relation, the *i*th bit in the bitmap is set to one. Since the tuples a transaction will reference vary widely, no attempt has been made to provide the ability to perform block updates of the read or write sets. Additionally, to minimize the probability that the data sets of two transactions will conflict, updates are recorded for each individual tuple. (In the parlance of lock-based concurrency control, "single-tuple granularity is used"). Although the updating of the data sets is somewhat costly due to its frequency, determining whether two data sets conflict is relatively inexpensive: the data set bitmaps are compared by performing a logical *AND*

on a word-by-word basis.

Since most of the concurrency control operations are relatively straightforward, they will not be discussed in detail. The Validate operation, however, is quite complex and warrants close scrutiny. The performance of the Validate operation is determined almost entirely by the Concurrency Control Manager's ability to efficiently compute the transaction's conflict set.

In the OPT-WAIT-50 scheme, the validating transaction, V, and an executing transaction, E, conflict "...iff the intersection of the write set of V and the current read set of E is non-empty." [Hari91 39]. StarBase employs several tricks to reduce the number of transactions and data sets involved in the conflict set determination. First, a data set, D, of V need only be compared if D's write set is non-empty. Second, D need only be compared to data sets which reference the same relation as it does. Finally, D need only be compared to data sets which have non-empty read sets. As discussed previously, determining data set intersection is fairly inexpensive since comparisons are performed on a word-by-word basis. Since each data set contains a referent to its transaction, the conflict set itself is easily determined from the conflicting data sets.

While Haritsa does not describe the implementation details of OPT-WAIT-50, he does give an implementation of a similar algorithm, OPT-WAIT [Hari91 144]. The implementation of OPT-WAIT requires that the conflict set need only be computed once per transaction and a count kept of the high priority transactions in the conflict set. Unfortunately this is not the case with OPT-WAIT-50: when a transaction commits it must know exactly which transactions must be aborted. In this case there are two design options: either a) compute the conflict set once and update it incrementally as transactions update their data sets or abort, or b) recompute the conflict set each time validation is retried. Since many of the cases in the option a) are difficult to determine, and the underlying assumption of any optimistic concurrency control algorithm is that the amount of conflicts will be small, StarBase implements option b). Whenever a transaction attempts to validate or aborts, every waiting transaction retries validation and will recompute its conflict set.

Note that in order to enforce the "current read set" stipulation, running transactions accessing a relation which the validating relation has accessed cannot be allowed to access the relation while the validator is computing its conflict set. All potential conflictors are suspended until the validator determines whether it must wait or commit.

VI. Improvements Over the Former Implementation

The new implementation of the StarBase front end is better than the previous for the following reasons:

1) The server has better control over the service order of transactions. In the old scheme, transactions with the same priority were processed in a FIFO manner--no consideration is given to the deadline. The new implementation processes transactions in a total ordering specified by the priority metric, which is a function of priority *and* deadline.

2) The server only needs to copy the transaction from the StarBase service port to its buffer of final destination once. The memcpy of the original implementation has been removed.

3) The StarBase transaction priority scheme is now completely independent of the number of Transaction Managers. In the previous implementation, priorities were limited to the number of Worker

threads in the server (a range of 0-7).

4) The new implementation avoids the dedicated Worker problem. For example, in the old implementation, if two transactions with the same priority *i* arrive, the second will have to wait for the server to complete the first transaction before it is processed. Worker *i* can only process transactions of priority *i*, so the second transaction must wait even if there are idle Workers of a different priority.

5) A Transaction Manager accepts a new transaction as soon as the Manager becomes ready rather than having new transactions assigned to it. The server does not have to perform any workload mediation; the next available Transaction Manager is guaranteed to service the next highest priority unserviced transaction in the server. If transactions were to be directly assigned to Transaction Managers and all Transaction Managers were busy, the next highest priority unserviced transaction could end up being assigned to a Transaction Manager which is running a very time-consuming transaction.

6) A mechanism now exists for StarBase to abort transactions in midstream. The previous implementation was not able to abort transactions, severely limiting its ability to enforce transaction deadline.

The new implementation of the StarBase Concurrency Control Manager is better than the previous for the following reasons:

1) The Concurrency Control Manager is better organized than the previous implementation. A well-defined interface for client transactions has been created. Inappropriately named data structures and functions now have names corresponding to actual items and processes in the concurrency control algorithm. Code is liberally commented and contains direct references to Haritsa's work [Hari91].

2) The new implementation attempts to minimize the amount of malloc'ing by allocating certain data structures in blocks. The previous implementation called malloc quite frequently to allocate relatively small amounts of memory. Although the new implementation has the potential to waste space, the heap manager will spend less time and space processing malloc requests.

3) The new implementation attempts to minimize the amount of searching required during validation. The previous implementation checked every data set of every waiting transaction for conflicts each time the conflict was recomputed. The current implementation cuts down this overhead using three tricks (see above), including checking only those transactions which reference a relation that the validator references.

4) The new implementation data set bitmaps are smaller than the old SET_LOCK bitmaps and are referenced more efficiently. Although the previous implementation attempted to allocate one bit per tuple in the bitmaps, compiler padding resulted in one byte per tuple allocations. Furthermore, the new implementation avoids the overhead of bit extraction when computing data set intersection by comparing bitmaps on a word-by-word basis. Furthermore, both bitmaps are only allocated if the transaction declares that it intends to read and write to the relation in question.

5) The new implementation preserves transaction serializability. In the previous implementation, transactions which could potentially conflict with a validator were allowed to continue updating their read sets during the commit.

References:

[Geor93] George, David W. “Implementation of Indexing and Concurrency Control Mechanisms in a Real-Time Database.” Technical Report. University of Virginia, 1993.

[Hari91] Haritsa, Jayant R. “Transaction Scheduling in Firm Real-Time Database Systems.” Technical Report 1036. University of Wisconsin, Madison, 1991.

[Lehm86a] Lehman, Tobin J., and Carey, Michael J. “Query Processing in Main Memory Database Management Systems.” Proceedings of the ACM SIGMOD Conference. May 1986

[Lehm86b] Lehman, Tobin J., and Carey, Michael J. “A Study of Index Structures for Main Memory Database Management Systems.” Proceedings of the Twelfth International Conference on Very Large Databases. August 1986.

[Loep92] Loepere, Keith. *Mach 3 Kernel Principles*. Open Software Foundation/Carnegie Mellon University 1992.

[Sava92] Savage, Stefan, and Tokuda, Hideyuki. “Real-Time Mach Timers: Exporting Time to the User.” Unpublished. Carnegie Mellon University, 1992.

[Son92] Son, Sang H., George, David W., and Kim, Young-Kuk. “Developing a Database System for Time-Critical Applications on RT-Mach.” Unpublished. University of Virginia, 1993.

[Toku90] Tokuda, Hideyuki, Nakajima, Tatsuo, and Rao, Prithvi. “Real-Time Mach: Towards Predictable Real-Time Systems.” Proceedings of the USENIX 1990 Mach Workshop, October 1990.

[Toku91] Tokuda, Hideyuki, and Nakajima, Tatsuo. “Evaluation of Real-Time Synchronization in Real-Time Mach.” Proceedings of the USENIX 1991 Mach Workshop, October 1991.

[Toku92] Tokuda, Hideyuki, Savage, Stefan, et al. “ART Tutorial on Real-Time Mach.” Slides. Carnegie Mellon University, 1992.

[Tute89] Tuten, Alan. “SDB--Simple Relational Database System: a User Guide.” Technical Report. University of Virginia, 1989.