An Object-Based Approach to Implementing Distributed Concurrency Control

Steven S. Popovich
Gail E. Kaiser
Shyhtsun F. Wu

Columbia University
Department of Computer Science
New York, NY 10027

CUCS-051-90
20 November 1990

Abstract

We have added distributed concurrency control to the MELD object system by representing in-progress transactions as simulated objects. *Transaction objects* exploit MELD's normal message passing facilities to support the concurrency control mechanism. We have completed the implementation of an optimistic mechanism using transaction objects and have designed a two-phase locking mechanism based on the same paradigm. We discuss the tradeoffs made and lessons learned, dealing both with transactions *on* objects and with transactions *as* objects.

Popovich and Wu are supported in part by the Center for Telecommunications Research. Kaiser is supported by National Science Foundation grants CCR-9000930, CDA-8920080 and CCR-8858029, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, SRA, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

topics: Languages, Tools and Software Engineering; Distributed Databases

An Object-Based Approach to Implementing Distributed Concurrency Control

Steven S. Popovich
Gail E. Kaiser
Shyhtsun F. Wu

Columbia University
Department of Computer Science
New York, NY 10027

20 November 1990

Abstract
We have added distributed concurrency control to the MELD object system by representing in-progress transactions as simulated objects. *Transaction objects* exploit MELD's normal message passing facilities to support the concurrency control mechanism. We have completed the implementation of an optimistic mechanism using transaction objects and have designed a two-phase locking mechanism based on the same paradigm. We discuss the tradeoffs made and lessons learned, dealing both with transactions *on* objects and with transactions *as* objects.

1. Introduction
The MELD distributed programming language provides the application programmer with *transaction blocks*, sequences of statements that are serializable with respect to all other MELD code. MELD application code, whether or not within a transaction block, may include both synchronous procedure calls and asynchronous messages to the same or other objects in the same or different processes, possibly executing on different hosts. All code executed within a transaction block and any code executed by threads created directly or indirectly by the block are part of the same transaction.

The novel aspect of MELD's implementation of transactions is that the various entities enforcing the concurrency control mechanism are represented as simulated ''objects'' that communicate using MELD's normal message passing facilities. This notion of *transaction objects* permits significant reuse of design and code in the implementation of concurrency control mechanisms, reusing design/code from other parts of the object system implementation to support concurrency control and permitting substitution of one concurrency control mechanism for another, reusing much of the same design/code. We have implemented an optimistic concurrency control mechanism reusing the existing interprocess communication facilities, and have designed a two-phase locking mechanism reusing the design for optimistic concurrency control.

The MELD distributed object system has been under development for several years, and a previous paper [8] presents the language and implementation as it was before transaction blocks were added. In a more recent paper [9], we describe the interactions between the three concurrency control policies now included in the language — serializable transactions for concurrency control across multiple objects residing in multiple processes, atomic blocks for critical sections on individual objects, and uncontrolled concurrency. This paper discusses our experience developing the first policy based on our notion of transaction objects.

We first give brief background information on the MELD object system, and then describe the optimistic concurrency control protocol we adapted to our transaction object paradigm. We explain the implementation of distributed optimistic concurrency control in terms of simulated objects and walk through a portion of a transaction processing demo program emphasizing the communication among these objects. We then discuss the experience gained and lessons learned through this effort. We go on to present our design for a distributed two-phase locking concurrency control protocol reusing the same interface, enabled by our transaction object paradigm. We briefly compare to related work, and conclude by sketching our ideas for future work.

2. Background: The MELD Object System

MELD is an experimental object system under development at Columbia University. The object-oriented programming language supports classes, strong typing of instance variables, active values, multiple inheritance, and separate compilation of modular units called *features* that bundle together related classes and objects. These facilities were developed for an early version of MELD [6, 7], without persistence, concurrency or distribution.

MELD now allows any class to be declared as persistent, and instances of persistent classes are stored in a B-tree database. Access to persistent and/or remote objects is through a network name service using a uniform syntax, so that application code need not ‘‘know’’ whether objects are stored locally or remotely, transiently or persistently. The MELD programming language is now concurrent, with new threads of control created by asynchronous messages and also effectively by synchronous procedure calls, which are implemented internally by an asynchronous message pass followed immediately in the sending thread by a statement waiting for the return message. Multiple threads may execute concurrently in the same object as well as in different objects.

MELD is implemented by translating the MELD code into C. The compilation system consists of a preprocessor `mpp` (which resolves inheritance), a compiler `mc` (which generates C code and feature symbol tables), and a linker `ml` (for combining separately compiled features). The runtime system consists of a kernel (supporting scheduling, persistence, transaction objects, *etc.*) linked into all MELD applications, and a separate `nameserver` program that must be running on every host where a MELD application executes. Arbitrary C code may be called from within MELD code, and is also linked by `ml`. The compilation system contains approximately 9600 lines of C, 3800 lines of Yacc, and 1100 lines of Lex. The runtime kernel includes approximately 11,300 lines of C, 700 lines of Yacc, and 200 lines of Lex (some of this is for the MELD Debugger [5], not discussed here). The name server consists of approximately 2400 lines of C.

3. Background: Concurrency Control Algorithms

Our first application of the transaction object concept implements the multi-version parallel validation (MVPV) optimistic concurrency control algorithm. MVPV, presented in a paper by Agrawal *et al.* [1], is a variant of the original optimistic concurrency control algorithm devised by Kung and Robinson [10]. Only short descriptions of these algorithms are included here; the full details are in the papers.[1] Our second application of transaction objects, only a design at this point, is to the conventional two-phase locking concurrency control algorithm (2PL) [2]. We assume that the reader is familiar with 2PL. Both MVPV and 2PL employ standard distributed commit protocols, involving a coordinator in the process where the transaction is initiated and cohorts in every process where the transaction executes; we also assume the reader is familiar with such commit protocols.

Optimistic concurrency control mechanisms are so named because they do not use locking to protect objects from concurrent access. Instead, they make the ‘‘optimistic’’ assumption that there will not be any conflicts among transactions, and then check when each transaction completes whether this assumption was in fact true. When a conflict is detected with an earlier transaction, the later transaction is rolled back and restarted; otherwise, it is committed.

Each optimistic transaction is implemented in three phases, a *read phase* where objects are read and shadow copies of objects are written, a *validation phase* where the runtime system checks whether or not the ‘‘optimistic’’ assumption was valid, and a *write phase* for validated transactions where the shadow updates are copied to the actual objects. During the read phase, a transaction accesses any resources it needs without waiting for locks, but instead records the sets of objects read and written (the transaction’s *read set* and *write set*). The transaction is *validated* when it finishes by comparing its read set and write set against other previously completed transactions. The read set must not conflict with the write set of any other transaction that validated since this transaction began, and the write set must not conflict with any other transaction that has entered validation but not yet completed its write phase.

A simple pseudo-code description of Kung and Robinson’s original optimistic algorithm is given in Figure 3-1. This version assumes that only one transaction is validating at any given time; their paper also presents an algorithm for validating transactions in parallel. The *tbegin* section is executed at the beginning of each transaction to initialize the transaction’s internal data. Then the transaction executes, and collects its read and write sets. The *tend* section runs when the transaction is ready to commit, and performs validation.

Kung and Robinson use parentheses "(...)" as begin-end blocks and angle brackets "<...>" as critical section delimiters, indicating the portion of the pseudo-code that must be executed serially with respect to other transactions. We have omitted portions of their algorithm having no impact on concurrency control.

The MVPV validation algorithm is based on two *transaction number counters*: a *commit* transaction number counter (*ctnc*) and a *visible* transaction number counter (*vtnc*). *ctnc* counts

---

[1] According to one of the authors of the MVPV paper, Soumitra Sengupta, ours is the only implementation of that algorithm.

```
tbegin =
      ( read set := empty;
        write set := empty;
        start tn := TNC )

tend =
      (< finish tn := TNC;
         valid := TRUE;
         FOR t FROM start tn+1 TO finish tn DO
             IF (write set of transaction with transaction number t
                 intersects read set)
             THEN valid := FALSE;
         IF valid
         THEN ((write phase); TNC:=TNC+1; tn:=TNC) >;
       IF valid THEN (cleanup)
       ELSE (backup) )
```

Figure 3-1: Basic Optimistic Concurrency Control Algorithm

```
InitialCS:
    < ctnc = ctnc + 1;
      tn(T) = ctnc;
      Allocate entry E(T);
      E(T).id = T;
      E(T).type = VALIDATING;
      E(T).num = tn(T);
      AQCopy(T) = CopyAQ([sn(T), tn(T)])
      InsertAQ(E(T), tn(T)); >

Validation:
    IF (ValidatePredecessors(T, [sn(T), tn(T)]) != NULL)
    THEN { Abort(T); EXIT }

WritePhase:
    Write(WS(T), tn(T));
    < E(T).type = WRITTEN;
      FOR E(Ti) IN AQ: vtnc < E(Ti).num <= ctnc IN INCREASING ORDER DO
        IF E(Ti).type = WRITTEN
        THEN vtnc = E(Ti).num;
        ELSE EXIT; >

FUNCTION ValidatePredecessors(T: TxnDesc; [from, to]: TxnNoInterval): TxnNo;
BEGIN
    FOR E(Ti) IN AQCopy(T): from < E(Ti).num <= to IN INCREASING ORDER DO
        IF WS(Ti) INTERSECT RS(T) != NULL_SET
        THEN { RETURN(E(Ti).num; }
        RETURN(NULL);
END ValidatePredecessors
```

Figure 3-2: Multi-Version Parallel Validation Algorithm

the transactions that have entered validation, while *vtnc* holds the highest transaction number whose results are visible to other transactions (i.e., its write phase has completed). This is the latest transaction that has completed its write phase and has no transaction before it that has not also finished. MVPV combines optimistic concurrency control with multiple versions [12], so read operations of a transaction access the last visible version written prior to the start of the transaction. Validation is thus unnecessary for read-only transactions, which obtain a consistent but not necessarily up-to-date view. For applications with many read-only transactions, MVPV has an advantage over pessimistic (locking) schemes as well as the original optimistic scheme, because of the very low overhead for read-only transactions. A pseudo-code description of the MVPV algorithm is given in Figure 3-2.

A transaction *T* begins (not shown) by being assigned a *start number* (*sn*(*T*)) that defines the highest transaction number whose results are visible to *T* (the current *vtnc*). As *T* continues its read phase, its read (*RS*(*T*)) and write (*WS*(*T*)) sets accumulate. Reads of data not yet written by *T* see the last version of the object written before *sn*(*T*), while writes create a new shadow version specific to *T* as in the original optimistic algorithm, which is then seen by subsequent reads by *T*. Many of these versions may exist at any time, since there can be one for each active transaction. There may also be more than one visible version of the same object; old versions must be kept until such time as there are no transactions that may reference them. Eventually *T* ends (from the application program's point of view), entering its validation phase.

The initial critical section, InitialCS, then assigns the transaction its *transaction number tn(T*) (the incremented *ctnc*), which determines the serialization order of transactions, and sets up the transaction's entry in *AQ*, the *active queue*. *AQ* contains an entry for every transaction that has entered validation prior to this one, for the transaction to validate against. Actually, each transaction keeps a partial copy, *AQcopy*, of *AQ* to avoid problems with concurrency on this important data structure as they validate in parallel. Entries in *AQ* older than the earliest *sn* of any running transaction will never be considered in validation, and so may be discarded. This keeps the space taken by *AQ* from growing without bound as long as all transactions eventually terminate.

After InitialCS completes, Validation proceeds concurrently with other activities. It checks that there is no transaction with a transaction number *tn* after *sn*(*T*) that has written a new version of any data read by *T*. (Concurrently validating transactions never cause ambiguities in this ordering, since *tn*'s are assigned serially by InitialCS.) If such a transaction exists, *T* must abort, since its results are based on old data and may not be consistent with the new data. This is unnecessary when the current transaction is read-only, because such a transaction may always be considered to have finished at its start time without affecting the serialization of other transactions. If *T* validates successfully and can commit, it enters its WritePhase, and a final critical section marks *T* as having written its results and updates `vtnc`. *T*'s new versions now become public, and are visible to new transactions if the new *vtnc* is at least *tn*(*T*). If not, they will become visible at a later time, when this condition becomes true.

We have presented the MVPV algorithm in centralized rather than distributed form; the distributed form is described in detail in Agrawal *et al.*'s paper. The transformation to the distributed form involves consideration of the *range* of start numbers and transaction numbers for different *cohorts*, or sites participating in the transaction. The range is determined by a *coordinator* by combining information from all the cohorts, and the validation algorithm is executed at each of the cohort sites of a transaction with conflicts detected over the full range rather than using any of the various local start and transaction numbers.

This process requires a three-phase commit protocol instead of the usual two-phase protocol; an additional phase is added at the beginning to calculate the range. The three phases are as follows:

1. *Determine global transaction numbers* — The coordinator detects termination of a transaction and asks each cohorts to send back its local numbers for the transaction. Each cohort replies with its local start number and local transaction number. The

coordinator determines the global start number and global transaction number by taking the minimum of the local start numbers and the maximum of the local transaction numbers (respectively).

2. *Validate* — The coordinator sends the global start and transaction numbers to all cohorts. Each cohort replies, informing the coordinator whether or not the transaction could be validated at the cohort with the global numbers.

3. *Commit or Abort* — If the transaction was validated at all cohorts, the coordinator tells the cohorts to commit, and they update their current versions of all objects involved in the transaction from their shadow versions of those objects. Otherwise, the coordinator tells the cohorts to abort, and they discard their shadow versions for the transaction.

This description has been somewhat simplified from Agrawal *et al.* to avoid discussing technical details such as the safety factor that is applied to the global transaction number. For a complete description, see their paper.

4. Implementation of Distributed Optimistic Concurrency Control

MELD's transaction subsystem implements the distributed form of the MVPV algorithm. The main technical decision that we made in adapting the algorithm was the granularity at which to apply it: to entire objects or to individual instance variables within objects. We opted for multi-version variables, rather than objects, because of the increased concurrency this allows. Ramifications of this decision are discussed in Section 6.

We represent transactions as C structures, as first-class MELD objects are also represented internally, but the C structures do not possess all the attributes of MELD object structures, and cannot be treated as MELD objects by the runtime system; thus we refer to transaction objects as ‘‘simulated’’, even though the same message passing facilities are employed. C declarations for the major data structures of the transaction implementation are shown in Figure 4-1.

A *transaction* structure may represent either a coordinator or a cohort object. Some fields are used only in coordinators, some only in cohorts, but most are used in both kinds of objects. Had we implemented transactions as first-class MELD objects, we would most likely have defined two subclasses of a "Transaction" class, namely "Coordinator" and "Cohort", rather than use only one data structure for both. Since transactions are not first-class MELD objects, but only C structure simulations, and we needed to be able to treat both coordinator and cohort objects as transactions without distinction in some cases, we employed a single structure.

A transaction object consists of:

- A **type** that distinguishes between ‘‘real’’ and ‘‘fake’’ transactions. MELD code initiated directly or indirectly by a transaction is part of the ‘‘real’’ transaction. All other MELD code, running outside any transaction, uses the notion of ‘‘fake’’ transactions to keep track of what it has read and written so that ‘‘real’’ transactions can validate against these operations. ‘‘Fake’’ transactions cannot abort, but they may cause ‘‘real’’ transactions to abort due to conflicts. The need for ‘‘fake’’ transactions is explained in Section 6.

- A global host (**g_host**) string that contains the name of the machine where the

```
struct meld_set {
  /* Details omitted; a list containing a set of pointers to variables. */ };

struct meld_process_id {
  char                *classname;
  char                *objname;
  int                  ps_no; };

struct transaction { /* simulated object */
  int                 type;
  char                *g_host;
  struct transaction *g_tp;
  struct meld_process_id  *g_ps_id;
  struct meld_process_id  *l_ps_id;
  int                 g_sn;
  int                 g_tn; /* == l_tnc */  /* will be final gtnc */
  int                 l_vtnc;               /* version index */
  int                 readonly;
  struct SEnd_pair    *send_pair;  /* the capitalization is intentional */
  struct visible_tn   *visible_tn;  /* only the coordinator will use it */
  struct meld_set     *read_set, *write_set;
  struct event        *_tr_ep;
  int                 _tr_occ; };

struct visible_tn {
  char                     *l_host;
  struct meld_process_id  *l_ps_id;
  struct transaction       *l_tp;
  int                       vtni;
  int                       l_tnc;
  struct visible_tn        *next; };

struct SEnd_pair {
  char                      *l_host;
  struct meld_process_id   *l_ps_id;
  struct SEnd_pair          *next; };
```

Figure 4-1: Transaction System Data Structures

coordinator of this transaction lives. This is the same in both the coordinator and the cohorts of a transaction. It exists only for historical reasons; it has been superseded by the **g_ps_id** field (see below).

- A global transaction pointer (**g_tp**) that points to the coordinator object in both the coordinator (where it points to itself) and the cohorts. This is not a valid pointer except in the process where the coordinator lives, but it serves as identification and as a means of distinguishing coordinator objects from cohort objects.

- A global process id (**g_ps_id**) and a local process id (**l_ps_id**) that record the object names that the coordinator (global) and cohort (local) objects, respectively, are registered under with the name server. Before an object can receive messages from remote objects, it must be registered; when messages are sent to the coordinator or a cohort, they are sent to these names. This structure and its use are described in more detail below.

- A global start number (**g_sn**), a global transaction number (**g_tn**), and a local visible transaction counter (**l_vtnc**) that serve as the variables of similar names in the distributed MVPV algorithm. **g_sn** is the minimum of all cohorts' local start numbers, and **g_tn** is the maximum of the local transaction numbers, disregarding safety factors. These two fields are used only by coordinators. **l_vtnc** is used by cohorts; it is the local start number for the transaction. Public versions created before **l_vtnc** are visible to the cohort.

- A **readonly** field that is true if the transaction has not written any data. In the cohorts, this field is not used; a cohort is read-only if its write set is empty. In the coordinator, this field summarizes the information in all the cohorts. When a read-only transaction terminates, no validation need be done.

- The send-end pair list (**send_pair**) that exists only in the coordinator and keeps track of all of the threads running within the transaction. Its job is termination detection; a simple algorithm is used for this. The name comes from the messages used in the algorithm, **send_begin_msg** and **send_end_msg**. Every time a MELD method is called within a transaction, creating a new thread, a **send_begin_msg** is sent to the coordinator. The coordinator then adds a **SEnd_pair** element to this list to represent the thread. Every time a method terminates within a transaction, a **send_end_msg** is sent, and the corresponding element is removed from its coordinator's list.[2] A transaction finishes when its **send_pair** list becomes empty, signifying that all methods (and thus all threads) within the transaction have terminated. The distributed commit protocol then begins.

- A visible transaction number structure list (**visible_tn**) that keeps an entry for each process that has a cohort in the transaction. Only coordinators have this field. It is used during the commit protocol.

- A **read_set** list and a **write_set** list that keep track of the instance variables that have been read and written, respectively, by the transaction within the local process. These are found only in cohorts.

- Other information (**_tr_ep** and **_tr_occ**) that is needed in case of an abort in order to restart the transaction, but the format is not relevant to this discussion. These are only in the coordinator, as individual cohorts cannot restart independently of the coordinator.

A **meld_process_id** is an address for a process in which a transaction object resides, as understood by the MELD name service. It consists of a *classname* and an *objname* unique within the class. The **meld_process_id** is used whenever it is necessary to send a interprocess message from one transaction object to another. It contains the classname and objname strings of an ‘‘object’’ representing the coordinator's process (in the **g_ps_id** field) or the local process (in the **l_ps_id** field). We use process identifiers, rather than individual transaction object identifiers, to avoid the overhead associated with notifying the name service of the creation and destruction of each transaction. In use, a **meld_process_id** is always paired with a transaction pointer (valid in its own process only). The name service delivers each message to its destination process, and within that process, the transaction pointer provides a local dispatch address. The classname is fixed for all transactions[3] , and the objname is a printable form of the process identifier on the local machine. The **ps_no** field holds the process identifier as a number. It is present for historical reasons and is no longer used, except in debugging.

---

[2]Actually, any element with the same **l_ps_id**, it does not matter unless a **send_end_msg** arrives before a matching **send_begin_msg**. The termination algorithm, designed for local area networks where that case does not occur, does not handle it well.

[3]Since a **meld_process_id** identifies a process to the name server, the word "SERVER" is used as the classname.

In MELD, all message passing outside of the local process goes through the name service, which looks up the name and forwards the message to the proper host by means of a server-to-server protocol, where it is delivered to the destination process by the remote server. This is true for both ordinary messages between first-class MELD objects and for the special messages between transactions; the only difference is in how they are treated in the network message handler once they arrive at the local MELD process. Non-transaction messages, including both ordinary and return messages, are delivered directly to the named object. (Return messages are the replies, usually containing a return value, sent back to the caller at the completion of a synchronous RPC.) Transaction messages are sent to the particular transaction ‘‘object’’ whose pointer (in the named process) is included in the message.

The **visible_tn** structure holds information that enables the coordinator to find each of its cohorts when it runs the commit protocol. The coordinator has a list, each element of which includes the local hostname of the cohort and its message passing address (its (**l_ps_id** plus its local transaction pointer). During the commit protocol, **validate_request** messages must be sent to all cohorts, **validate_return** messages received from them, and either **real_write** or **destroy_version** messages sent back, depending on whether the transaction commits or aborts.

In addition to the necessary addresses, the **visible_tn** structure also contains information about the transaction number variables maintained by the MVPV algorithm at each site. These are used to derive the global start number and global transaction number of the transaction: a multi-process transaction is considered to have started at the earliest of its local start numbers (**vtni**), and to have entered validation at the latest of its local transaction numbers (**l_tnc**), plus a small safety factor. (**g_sn** holds this earliest start number, and **g_tn** holds the latest transaction number (including the safety factor), in the **transaction** structure.) This leads to unnecessary aborts in situations where the current and visible transaction numbers are not in reasonably close synchronization among cohorts. We have worked out a solution where, whenever a transaction finishes, these variables are synchronized in all of the participating processes in the transaction. We describe the problem and its solution in detail in Section 6.

Figure 4-2 gives the headers and brief descriptions of each of the operations (‘‘methods’’) on transactions. All operations are implemented as C functions, with local optimistic concurrency control-related operations on cohort objects called by compiler-generated C code within the local process, and global distributed commit-related operations on coordinator and cohort objects invoked by sending the appropriate messages using MELD’s normal interprocess message passing facilities.

5. Example of MELD Transaction Management

We now show how transaction objects are used in MELD by walking through an example transaction. Our example comes from a toy stock trading application, with several distinct types of MELD objects: stocks, stock portfolios, checking accounts, and users. To buy a stock, a user sends a `buy` message to the user’s stock portfolio. This method, implemented as a transaction block, sends a `quote` message to the stock to get its current price and a `withdraw` message to the user’s checking account to get the money to buy the stock. It then updates the user’s portfolio to reflect the stock that was purchased.

Local Optimistic Concurrency Control Operations

`do_read(var, T)`
: Add `var` to read set of `T`, the transaction object.

`pre_read(T, var, value, size)`
: Read `T`'s version of `var` into memory pointed to by `value` with length `size`.

`do_write(var, T)`
: Add `var` to write set of `T`.

`pre_write(T, var, value, size)`
: Write `T`'s version of `var` from memory pointed to by `value` with length `size`.

`destroy_transaction(T)`
: Frees resources that are not needed after `T` finishes. Called by the `real_write` and `destroy_version` operations.

These misleading names exist for historical reasons. For example, `do_write` *prepares for* `pre_write` *and is called before it.*

Global Distributed Commit Protocol Operations

`send_begin_msg()`
: Notifies the coordinator that a method has begun execution within `$self`, the transaction object.

`send_end_msg(tnc, vtnc, readonly)`
: Notifies the coordinator that a method has completed execution within `$self`, and informs it of the current transaction number counter `tnc`, visible transaction number counter `vtnc`, and whether or not `$self` is `readonly` at the cohort process.

`begin_validate()`
: Starts validation of `$self` at coordinator; sends **validate_request** messages to cohorts.

`validate_request(g_sn, g_tn)`
: Asks a cohort to validate `$self` in its process with the specified `g_sn` and `g_tn`.

`validate_return(isvalid)`
: Returns VALID or INVALID to coordinator.

`real_write(wasvalid)`
: Writes `$self`'s versions of all variables to public versions if `wasvalid` is true. In the current MELD system, `wasvalid` is always true when `real_write` is called.

`destroy_version()`
: Discards `$self`'s versions of all variables.

Figure 4-2: Operations on Transaction Objects

This activity thus involves three objects, which must be kept consistent, and up to three cohorts since the objects may reside in different processes (which may execute on different hosts), as depicted in Figure 5-2. Figure 5-1 shows the relevant portion of the MELD code for these methods. Due to space limitations, the code for demo-related messages has been removed, leaving only the basic checking account and stock manipulation code.

We assume that the user already owns some shares of IBM and wants to buy 10 more. To do this, the user types a message on the terminal that causes the user object to send the message `buy(10, "IBM")`] to the corresponding stock portfolio. The events from this point onward are depicted in Figure 5-2; the numbers below correspond to those on the arrows in the diagram.

1. The `buy` message arrives at the portfolio object and the corresponding method is invoked. Entering the transaction block (marked by "<<...>>" in the code) has the side effects of creating a coordinator object and a cohort object in the current process, and then the cohort sends the **send_begin_msg** to the coordinator. (In this example, names and code *written in* MELD appear in `typewriter font`, and

```
PERSISTENT CLASS ChkAcct ::=
  balance: real;
METHODS:
  withdraw(money: real; send_obj: name): boolean --> ( /* atomic block */
    if (balance < money)
      then [
        return(false); /* fail -- non-sufficient funds */
      ]
    else [
      balance := balance - money;
      return(true);
    ]
  )
END CLASS ChkAcct

PERSISTENT CLASS Portfolio ::=
  stockNames: array[0..4] of string;
  stockHoldings: array[0..4] of integer;
METHODS:
  buy(BuyName: string; buyNumShares: integer; acct: ChkAct) --> <<
    stk: Stock;
    i: integer;
    stkPrice: integer;

    stk := Stock.get(BuyName);
    if (stk != nil)
      then [
        stkPrice := stk.quote();
        if (acct.withdraw((real)stkPrice * buyNumShares / 8, $sender))
          then [
            i := $self.findStock(BuyName);
            if (i < 0)
              then [ /* add a new stock to the portfolio */
                i := $self.findSpace();
                $self.setHoldings(i, buyNumShares, BuyName);
              ]
              else [ /* buy more of a previously held stock */
                $self.updateHoldings(i, buyNumShares);
              ]
          ]
          /* else fail -- not enough funds to buy stock */
      ]
      /* else fail -- cannot find the named stock */
  >>
END CLASS Portfolio
```

Figure 5-1: MELD Transaction Example: Code

names and code *implementing* MELD appear in **boldface**. In figures, all code appears in `typewriter font.`) When the **send_begin_msg** is received by the coordinator, an element is added to its **send_end_pair** and **visible_tn** lists to represent this thread of control.

2. The portfolio then gets a handle on the persistent object representing the stock named `"IBM"`, which involves an exchange of messages with the name server (not shown in the diagram), and sends a `quote` message to the stock. This message implicitly carries the transaction identifier, which is the address of the coordinator object (**meld_process_id**). When the `quote` method is invoked, this has the side effects of creating a local cohort for the transaction in the stock object's process and sending a **send_begin_msg** message back to the coordinator. Again, when this is received by the coordinator, an element for this thread is added to the **send_end_pair** and **visible_tn** lists. (Actually, there is only one element in the **visible_tn** list per cohort rather than per thread, but there is no distinction in this

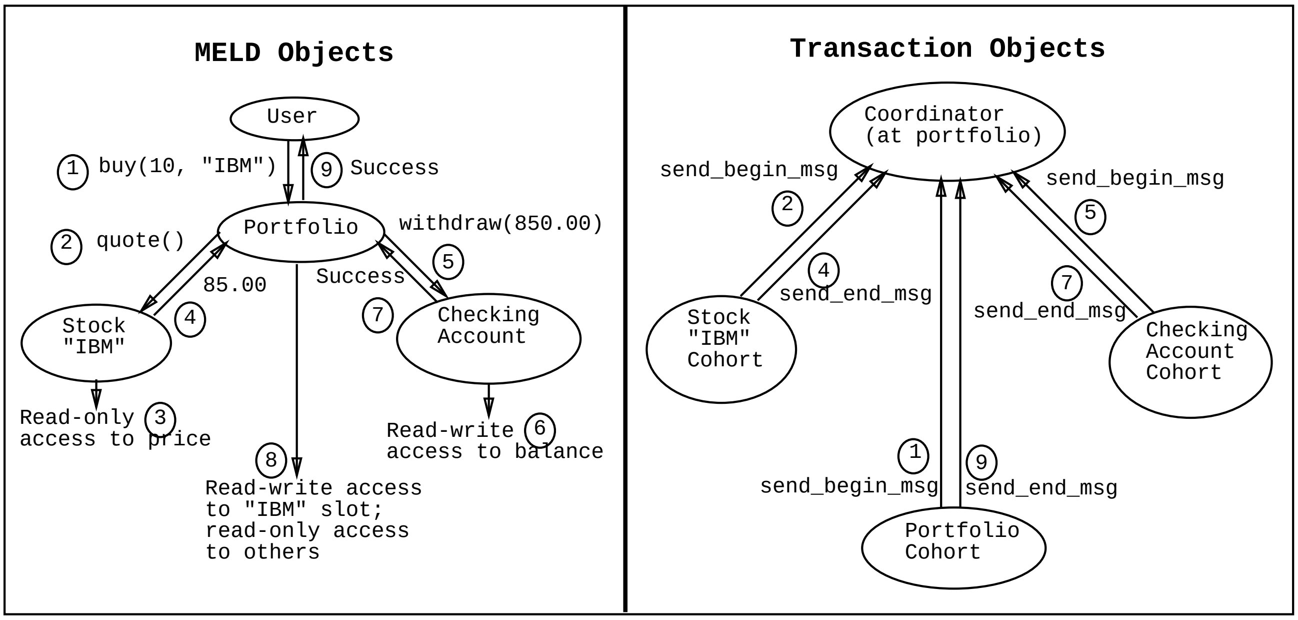

Figure 5-2: MELD Transaction Example: `buy(10, "IBM")`

example.)

3. When the `quote` method (not shown) reads the `price` instance variable of the stock object, the **do_read** operation is invoked to add this variable to the cohort's **read_set** and then the **pre_read** message accesses the appropriate version of this instance variable (the version written by the last visible transaction before the start time of the cohort, its **vtnc**).

4. The `quote` method sends a message returning the stock's price per share (for example, $85.00) back to its caller, the stock portfolio. As a side effect, a **send_end_msg** message to the coordinator is also generated; this lets the coordinator know that the thread in the stock object has completed, which is recorded by removing the corresponding **send_pair** element from the **send_end_pair** list (but the element corresponding to the stock object's cohort remains in the **visible_tn** list). When the return message arrives, the `buy` method in the portfolio object continues.

5. The next step is to withdraw the money to buy the stock, so `buy` sends a `withdraw` message to the checking account object. When the `withdraw` method is invoked, it has the side effects of creating a cohort object in the checking account's process, since the transaction has not previously involved that process, and sends a **send_begin_msg** to the coordinator object. The `withdraw` method checks that the balance is sufficient; in doing this, the **do_read** and **pre_read** operations add the `balance` instance variable to the cohort's **read_set** and read the appropriate version of the `balance` value for the transaction, respectively.

6. `withdraw` then subtracts the amount specified from the balance and writes the new balance back; the side effects are **do_read**, **do_write** and **pre_write** operations for `balance`. The **do_read** does nothing, since `balance` is already in the **read_set**, **do_write** adds `balance` to the **write_set**, and **pre_write** creates the appropriate new shadow version for this instance variable.

7. The `withdraw` method concludes by sending a message back to the portfolio indicating that the withdrawal succeeded, and the `buy` method continues again. This return message has the usual side effect of a **send_end_msg** message to the coordinator, to tell it to remove this thread from its **send_end_pair** list.

8. The `buy` method next finds the slot for `"IBM"` in its portfolio and increments the number of shares by 10. This involves the four optimistic concurrency control transaction operations for adding the number of shares to the read set, actually reading the number of shares, adding this variable to the write set and writing its shadow version. Since the `"IBM"` slot is found via direct linear search, the variables (array elements) holding the names of all of the stocks before `"IBM"` in the portfolio were also read, with corresponding **do_read** and **pre_read** operations invoked as side effects.

9. Finally, when `buy` is finished, it sends a return message to the user object to say that it succeeded; this results in a **send_end_msg** to the coordinator. After processing this message, the coordinator realizes it has no more active threads, and so begins the commit protocol.

MELD's two-phase commit protocol resembles the last two phases of Agrawal's three-phase protocol, and is implemented by coordinator and cohort objects as follows:

1. The coordinator determines the global start number **g_sn** and global transaction number **g_tn** by taking the minimum of the local start numbers and the maximum of the local transaction numbers, plus a small constant safety factor. These numbers have already been collected, as the **vtni** and **l_tnc**, respectively, of each cohort represented in the **visible_tn** list. **vtni** was set by the earliest **send_begin_msg** message from the cohort, and **l_tnc** by the latest **send_end_msg** from the same cohort process (these values are piggybacked on all the termination detection messages from cohorts).

2. The coordinator sends this pair of numbers to all cohort objects in **validate_request** messages.

3. The cohorts reply with **validate_return** messages, which tell the coordinator whether or not the transaction could be validated at the cohort with these global transaction numbers.

4. If the transaction was valid at all cohorts, the coordinator then sends the cohorts a **real_write** message, and they update their current versions of all objects involved in the transaction from their shadow versions of those objects.

5. Otherwise, the coordinator sends the cohorts a **destroy_version** message, and they discard their shadow versions for the transaction.

To support recoverability, acknowledgments would be required from every cohort to complete the second phase, but this is not addressed in this paper. The commit protocol messages for our example are shown in Figure 5-3.

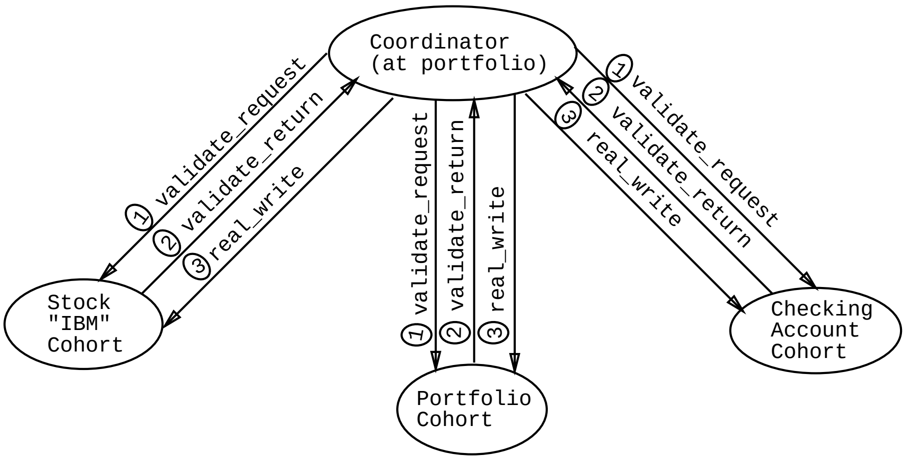

Figure 5-3: MELD Transaction Example Commit Protocol

6. Experience with Transactions *on* Objects

In Section 4, we touched on a problem caused by running an optimistic concurrency control algorithm across processes with widely differing usage patterns. Here, we describe that problem in detail and discuss several other important properties of transactions *on* objects and the difficulties they caused in the MELD implementation.

Consider the case where the participants in a transaction are a server and a client using the service, *e.g.*, a "bank" process providing access to checking account objects and a "user" process accessing a particular checking account. In this case, since the client only occasionally initiates a transaction with the server but the server may continually be responding to many clients' requests, it is easy for the server's transaction numbers to become much larger than the client's.

For example, a client and a server may have completed a transaction T_1, after which the client became inactive for a long period. This situation is shown in Figure 6-1. When the client initiates another transaction, T_2, with the server at time T_{2S}, the client's transaction number counter is be only 18 while the server's is 4096. If T_2 begins validation at T_{2V} after a small increase in the transaction numbers at both sites (bringing the client's transaction number to 19 and the busier server's transaction number to 4100), the validation algorithm may find a conflict between T_2 and some earlier T_1, although no conflict actually exists. This is due to taking the global start number as the earliest (18) of the local start numbers and the global transaction number as the latest (4100) of the local transaction numbers, plus a safety factor (omitted here and in Figure 6-1), and validating against this wide range of transaction numbers at all cohort sites. The validation algorithm sees the older transaction T_1 on the same account at the server as being in conflict with the more recent T_2, although they actually executed in serial order. This perceived conflict appears because the server's T_{1V}, 1029, is within the distributed validation range [18-4100].

Our solution to the problem of gross desynchronization of the transaction number counters exploits the transaction number information carried by **validate_request** messages from the coordinator to cohorts during the commit protocol. Instead of simply validating against the **g_sn**

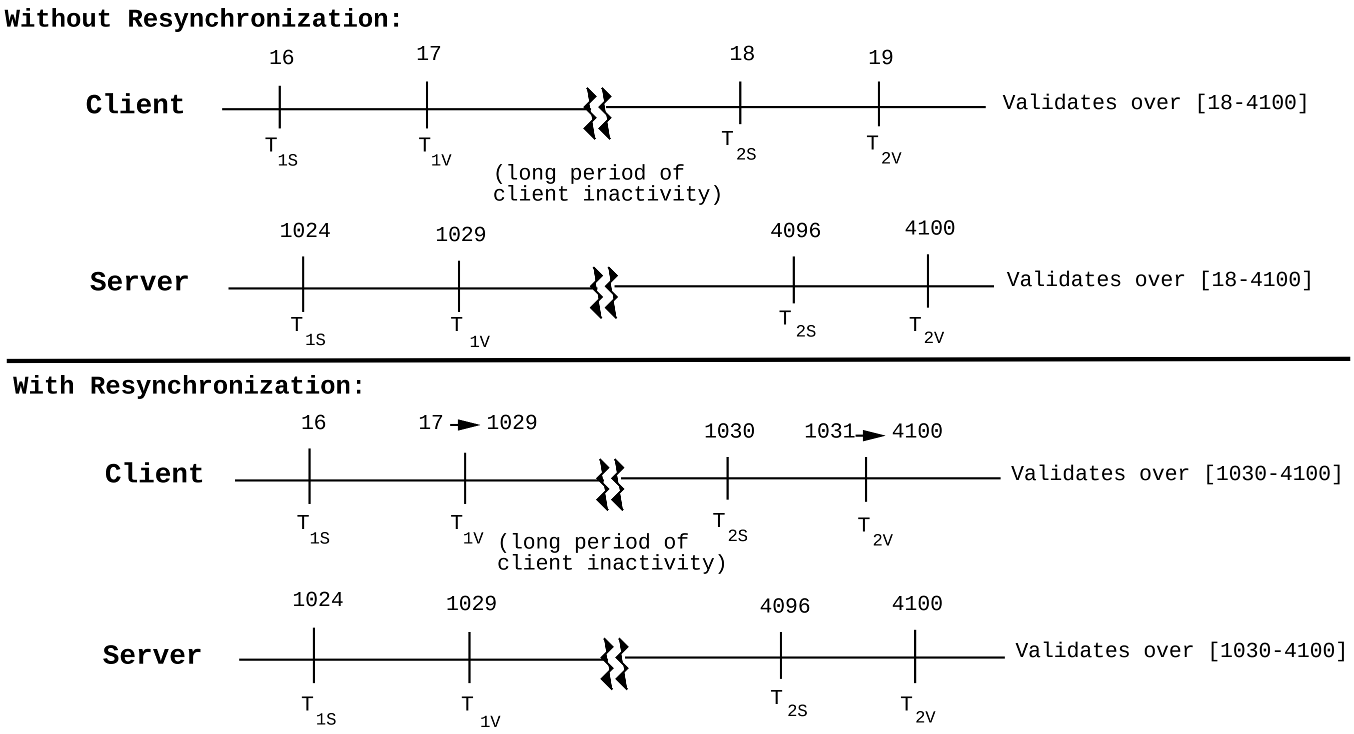

Figure 6-1: Example of Transaction Number Desynchronization

and **g_tn** provided, each cohort also updates its transaction number counter **ctnc** to the maximum of **g_tn** and the current value. As the second part of Figure 6-1 shows, this results in T_2 validating against the range [1030-4100] while the server's T_{1V} remains 1029, so the server no longer (erroneously) detects a conflict between T_1 and T_2.

This approach does not entirely prevent the problem of desynchronization, since a long-inactive client would remain out of synchronization until such time as it attempts to validate a transaction, and in fact, such a transaction is likely to abort due to being out of synchronization. But earlier transactions from the same client are guaranteed not to cause validation conflicts, since the client's and the server's transaction numbers were in agreement after the previous transaction, and in any case, even if the first transaction from an inactive client does abort unnecessarily, it will also synchronize the client with the server so that when the transaction restarts, the new incarnation will not encounter the same problem.

Subsequent transactions will remain reasonably well synchronized until the next long inactive interval, since the participants resynchronize at the end of each transaction. To prevent the problem entirely, it would be necessary to update *all* processes' local transaction number counters (**ctnc**'s) at the end of every transaction; this would not only be prohibitively expensive in a large system with many processes, but is impossible to guarantee in a distributed environment where new processes may be created on any host at any time. Note that the reason for synchronizing transaction numbers at the end rather than the beginning of each transaction is that the complete set of participants in a particular transaction is not known until termination is detected and validation begins.

In addition to the desynchronization problem, we discovered several other problems that occurred in the MELD implementation due to simply having transactions running *on* objects, rather than to treating the transactions *as* objects themselves. Assumptions that had implicitly been made during the initial implementation of MELD as a non-transactional object system were broken by the addition of transactions. For example, in our original implementation, all messages were treated identically. There was no difference between synchronous procedure calls and asynchronous messages; the compiler simply generated additional code to wait for the return message in the first case, but not the second. There was also no difference between these return messages and any others. It seemed like a nice, clean design.

When we added transactions to the MELD system, the message passing code was modified to transmit the current transaction identifier (the **meld_process_id** of the coordinator's process, plus a pointer to the coordinator ‘‘object’’ in that process) along with the message, and to create a cohort for the transaction in the remote process if one did not exist already, extending the transaction to that process. This caused a very hard-to-find bug, where if a method written as a transaction were called synchronously, later changes to data in the calling method would fail for no obvious reason. The bug turned out to be due to our uniform treatment of messages; the return message from the synchronously-called transaction was propagating the transaction identifier back to its caller and creating a cohort for it there, after the transaction had already "terminated" and started its commit protocol. Once this bug was found, fixing it was easy, though at the cost of treating return messages specially.

Another problem with transactions *on* objects deals with the treatment of interactions between transactional and non-transactional code on the data in the objects. In systems like Avalon [4], entire objects are declared as ‘‘atomic’’, meaning that every call to one of the object's methods is a separate transaction. In systems that support transactions across multiple objects, this problem must be dealt with somehow. Since concurrency control algorithms do not generally consider the possibility of non-transactional access to shared objects, that is, access by code that does not follow the conventions of the concurrency control mechanism. Non-transactional access is desirable in cases where performance is more important than strict data consistency, *e.g.*, usage statistics kept by a network management application [11].

In a system containing both transactions and non-transactional code, however, the non-transactional code must respect some of the transactions' concurrency control conventions. If the transactions use a locking algorithm, all code must acquire a lock before accessing an object for writing, or the transactions' integrity may be compromised. For reading, on the other hand, no lock need be obtained by non-transactional code. Without a read lock, the non-transactional code may see an inconsistent version of the data, but it cannot violate the consistency of the transactions as it could by writing without a lock.

In an optimistic algorithm, any code that fails to keep at least a write set for validation purposes may cause the same problem. Our solution was to create ‘‘fake’’ transactions for non-transactional code. The transaction objects for ‘‘fake’’ transactions look like any others except for their **type** field, and accumulate read and write sets in the same way as the ‘‘real’’ transactions do. They are not, however, validated when they terminate; their results are written to the actual objects (not shadow copies) without regard to any conflicts that may have occurred. Thus, non-transactional code may still corrupt other non-transactional code arbitrarily; we make no guarantees about that. What ‘‘fake’’ transactions do is allow ‘‘real’’ transactions to validate against *them* and abort (and restart) in case of conflict. This protects the ‘‘real’’ transactions against having their data corrupted by concurrent non-transactional code, while ‘‘fake’’ transactions still have a performance advantage, since they require no validation. This interoperability among MELD’s three concurrency control policies (no control, atomic blocks and transactions) is explored in detail elsewhere [9].

A final problem discovered during our experience implementing transactions *on* objects is in dealing with the creation of objects by transactions. Object creation is currently *independent* of transactions; that is, when an object is created within a transaction, it is created immediately, and may appear in the name space before the transaction commits. In the case that a transaction creates an object and aborts, the object is not deleted, but remains instead as a "content-free" object, with all of its variables set to null values.

This problem is a side effect of our decision to treat instance variables as the granularity of detecting conflicts among concurrent transactions, to allow greater concurrency than would be possible if entire objects were considered as units by the concurrency control algorithm. Since MELD has no versions of objects, but only of instance variables within each object, there was no place to put a notation that the object should appear in the name space only during the transaction’s write phase, or that it should be deleted if the transaction were to abort. This problem can obviously be solved by *ad hoc* means.

7. Experience with Transactions *as* Objects

As expected, object-based implementation of transactions permits significant code reuse. Our implementation successfully reused MELD’s name service and interprocess message passing facility. About 160 lines of message passing interface code were adapted to reuse ‘‘as is’’ approximately 800 lines of runtime support code and the 2400 lines of the name server in passing messages between transaction ‘‘objects’’. Implementing transactions as first-class MELD objects, rather than as simulated ‘‘objects’’ and ‘‘methods’’ (the C transaction structures and associated operations that we actually used), would most likely permit additional reuse of the object management and method invocation code.

We have also found that transaction objects permit design as well as code reuse. We have generalized the framework of transaction objects, and their local concurrency control and global distributed commit protocol operations, to support the standard two-phase locking protocol.

In our preliminary design, transactions continue to be represented as simulated ‘‘objects’’ by C structures with essentially the same local and global operations as for the optimistic concurrency control algorithm. The data structures for two-phase locking closely resemble those already implemented for the optimistic algorithm, and the changes are mainly confined to the concurrency control mechanism (local operations) rather than the commit protocol (global operations). To avoid having to add lock data to individual objects, replacing the multiple versions of instance variables for the optimistic mechanism, the preliminary design uses a separate lock manager ‘‘object’’ for each process. The relevant data structures are shown in Figure 7-1.

Because of this decision, no fields need be added to transaction objects to implement two-phase locking, *per se*; a locking transaction structure can be more or less the same as an optimistic transaction structure, with the optimistic concurrency control fields (**g_sn** through **write_set** in Figure 4-1) deleted. In our design, we have added one field, **bk_versions**; this is not necessary for concurrency control, but allows recovery in case a deadlock is detected and the transaction must be aborted. The **bk_versions** field contains a list of pairs of changed variables and their old values; in case of an abort, the variables would be restored to their backup values.

The lock manager in the preliminary design maintains two tables, one organized by object and searched to determine whether a lock exists on a particular object (or instance variable, depending on the granularity selected), and one organized by transaction and used at commit or abort time to free all locks belonging to the transaction. The lock table is represented as a structure holding these two lists. In a realistic implementation, these tables should not actually be linked lists, but should be implemented more efficiently. Here, however, we treat them as (separate) lists, since we are concerned only with the design and not the details of an implementation. The transaction and lock table data structures for two-phase locking are shown in Figure 7-1.

```
struct meld_process_id {
  char              *classname;
  char              *objname;
  int                ps_no; };

struct transaction { /* simulated object */
  int               type;
  char              *g_host;
  struct transaction *g_tp;
  struct meld_process_id  *g_ps_id;
  struct meld_process_id  *l_ps_id;
  struct SEnd_pair   *send_pair;  /* the capitalization is intentional */
  struct meld_version *bk_versions; /* backup version list, not shown */
  struct event       *_tr_ep;
  int               _tr_occ; };

struct lock_table { /* simulated object */
  struct obj_locks *objl;
  struct xact_locks *xactl; };

struct obj_locks {
  struct object *obj;
  struct obj_lock *locks;
  struct obj_locks *next_olocks; };

struct obj_lock {
  struct transaction *xact;
  int lock_type; /* actually an enumerated type */
  struct obj_lock *next_olock; };

struct xact_locks {
  struct transaction *xact;
  struct xact_lock *locks;
  struct xact_locks *next_xlocks; };

struct xact_lock {
  struct object *obj;
  int lock_type; /* actually an enumerated type */
  struct xact_lock *next_xlock; };
```

Figure 7-1: Locking Transaction System Data Structures

There are three local concurrency control operations for two-phase locking transactions: Obtain a read lock, obtain a write lock, and release all locks belonging to a transaction. The lock manager provides these operations, but they are routed through the transaction ‘‘object’’ to provide a uniform transaction interface and to allow the transaction to keep track of changes in its **bk_versions** list. Figure 7-2 shows the local and global operations for two-phase locking, using the same names as the optimistic algorithm and similar interfaces wherever possible. One local operation, to clean up the locks left at a cohort process by a transaction, has been added; two, for version maintenance of instance variables for the optimistic concurrency control algorithm, have been removed. (In a multi-version locking scheme, these two operations would return; however, since our initial implementation will not use multiple versions, these local operations are not needed.)

Local Two-Phase Locking Operations

`do_read(var, T)`
: Acquire a read lock on `var` for `T`. This is called **do_read** here because it will have calls to it generated at the same times and in the same way as the optimistic **do_read** operation is called in the current system.

`do_write(var, T)`
: Acquire a write lock on `var` for `T`. The value of `var` is also copied to a backup version stored in `T`. Like do_read, this operation simply replaces one of the optimistic protocol’s local operations.

`destroy_transaction(T)`
: Releases all locks held by `T`. Called by the two-phase locking `real_write` and `destroy_version` operations.

Global Distributed Commit Protocol Operations

`send_begin_msg()`
: Notifies the coordinator that a method has begun execution within `$self`. This and `send_end_msg` are the same as in the optimistic protocol, except that they no longer manipulate **visible_tn** structures.

`send_end_msg()`
: Notifies the coordinator that a method has completed execution within `$self`.

`begin_validate()`
: Starts validation of `$self` at coordinator; sends **validate_request** messages to cohorts.

`validate_request()`
: Asks a cohort whether or not it can commit `$self`.

`validate_return(isvalid)`
: Returns VALID or INVALID to coordinator. VALID means "ready to commit"; INVALID means `$self` must abort.

`real_write(wasvalid)`
: Conceptually "writes" `$self`’s versions of all variables to public versions if `wasvalid` is true. In the two-phase locking design, there is no version list as in the optimistic protocol. Instead, each transaction keeps a single backup version of its variables; so, `real_write` actually destroys this backup version and releases `$self`’s locks.

`destroy_version()`
: Conceptually "discards" `$self`’s versions of all variables. As with `real_write`, the name is now somewhat misleading. This operation actually writes the backup versions held by `$self` into the "real" objects.

Figure 7-2: Operations for Two-Phase Locking

The interface to the global operations of termination detection and the commit protocol remain the same except for the dropping of some now unnecessary arguments. The message types sent also remain the same, and in the same order, as in Figures 5-2 and 5-3, although their names are now less descriptive. Their implementations change as described in Figure 7-2. This design allows two-phase locking to be substituted for MELD's optimistic protocol, without any other changes to MELD's implementation.

8. Related Work

A few innovative engineering- or design-oriented systems represent development tasks as objects (*e.g.*, [3]). The purpose of these objects is to provide a persistent developer-visible "audit trail" of changes to the system under development and allow reasoning about these changes. Most of these systems are not concerned with reuse of concurrency control design or implementation, and in many cases are not concerned with concurrency control at all.

One notable exception is Cosmos [13], an operating system facility intended to support multi-user software development environments. Cosmos provides an explicit notion of *transaction objects*, which represent in-progress development tasks. A transaction object includes information about the task such as the new versions of changed development objects created during the task, and when committed turns into a persistent *configuration object* that represents a consistent state of the system under development. Concurrency control and recovery in Cosmos are implemented in terms of these transaction objects and configuration objects, in that a transaction object collects the new versions private to the development task until it is completed, and it is always possible to recover back to a previous configuration object. But there is no notion analogous to MELD's message passing among the objects in order to *implement* these mechanisms.

Avalon does use message passing within an object to implement concurrency control on that object, although there is no notion analogous to Cosmos's and MELD's objects representing transactions on *other* objects. Avalon provides a set of system-defined superclasses whose methods represent primitive operations for constructing synchronization and recovery mechanisms. This gives the programmer an object-oriented facility for defining new classes of objects with desirable properties, such as atomicity and persistence.

9. Conclusions

Our implementation of distributed concurrency control for MELD has provided some insights into the problems of transactions *on* objects, specifically their interactions with and impact on the message passing facility, concurrent non-transactional code, and object management operations. More significantly, treating transactions *as* primitive ‘‘objects’’ has allowed us to reuse some internal MELD facilities in the implementation of the transaction subsystem, and promises to allow reuse of design in implementations of alternative concurrency control mechanisms. There do not seem to be any particular aspects of our implementation of transaction ‘‘objects’’ that are peculiar to MELD, so this technique should also be useful for other distributed object systems.

Using transaction objects in our implementation enabled us to reuse MELD's existing communication mechanisms, which account for a significant fraction of MELD's runtime code. In our design for implementing an alternative concurrency control mechanism, two-phase locking (discussed in Section 7), we have reused and generalized the design for the transaction object mechanism. Transaction objects show great promise as an implementation strategy for reusable, extensible concurrency control in object systems.

Acknowledgments

Wenwey Hseush worked with the authors on the design and implementation of the previous version of concurrent MELD, without transaction blocks. David Staub developed MELD's name service. Quoc-Bao Nguyen ported MELD from Sun-3 under SunOS 4.0 to Sun-4, IBM RT under AIX 2.2.1 and Mach 2.5, and DecStation 3100 under Ultrix 3.1. David Garlan worked with Gail Kaiser on the initial development of a sequential version of the MELD language. We would like to thank Travis Winfrey for his comments on an earlier draft of this paper.

References

[1] D. Agrawal, A.J. Bernstein, P. Gupta and S. Sengupta.
Distributed Optimistic Concurrency Control with Reduced Rollback.
Journal of Distributed Computing 2(1):45-59, April, 1987.

[2] Philip A. Bernstein, Vassos Hadzilacos and Nathan Goodman.
Concurrency Control and Recovery in Database Systems.
Addison-Wesley, Reading MA, 1987.

[3] Don Cohen and K. Narayanaswamy.
A Logical Framework for Cooperative Software Development.
In *6th International Software Process Workshop*. October, 1990.
Preprints.

[4] David Detlefs, Maurice Herlihy and Jeannette Wing.
Inheritance of Synchronization and Recovery Properties in Avalon/C++.
Computer 21(12):57-69, December, 1988.

[5] Wenwey Hseush and Gail E. Kaiser.
Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language.
In *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 236-246. Madison WI, May, 1988.

[6] Gail E. Kaiser and David Garlan.
Melding Software Systems from Reusable Building Blocks.
IEEE Software 4(4):17-24, July, 1987.

[7] Gail E. Kaiser and David Garlan.
MELDing Data Flow and Object-Oriented Programming.
In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 254-267. Orlando FL, October, 1987.

[8] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush and Shyhtsun Felix Wu.
MELDing Multiple Granularities of Parallelism.
In *3rd European Conference on Object-Oriented Programming*, pages 147-166. Cambridge University Press, Nottingham, UK, July, 1989.

[9] Gail E. Kaiser, Wenwey Hseush, Steven S. Popovich and Shyhtsun F. Wu.
Multiple Concurrency Control Policies in an Object-Oriented Programming System.
In *Second IEEE Symposium on Parallel and Distributed Processing*. Dallas TX, December, 1990.
In press.

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

[11] Subrata Mazumdar and Aurel A. Lazar.
Knowledge-Based Monitoring of Integrated Networks.
In Branislav Meandzija and Jil Westcott (editors), *IFIP TC 6/WG 6.6 Symposium on Integrated Network Management*, pages 235-243. North-Holland, Boston MA, May, 1989.

[12] David P. Reed.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, MIT, September, 1978.
MIT LCS TR-205.

[13] J. Walpole, G.S. Blair, J. Malik and J.R. Nicol.
Maintaining Consistency in Distributed Software Engineering Environments.
In *8th International Conference on Distributed Computing Systems*, pages 418-425. San Jose, June, 1988.