The Replicated Object Store Service†

Barry Brachman and Gerald Neufeld
brachman@cs.ubc.ca, *neufeld@cs.ubc.ca*

Dept. of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z2

DRAFT – Do Not Distribute

July 9, 1995

Abstract

This paper describes the design of an application-independent, distributed and replicated object store service called *ROSS*. *ROSS* provides a high performance, highly-available object storage facility, providing both strong and weak object consistency. The former is realized using the conventional virtual partition algorithm while the latter is obtained by taking advantage of the characteristics of a common class of database applications where there is limited interdependency between objects. A distributed name service is one such application. The principal novelty of the approach is the notion of weak consistency, realized by a two-tiered transaction mechanism.

1 Introduction

The distributed, replicated object store service (*ROSS*) provides a highly-available, application-independent object storage facility. Each copy of an object store, called a *replica*, has its own database of objects. *ROSS* provides the programmer with a simple interface to atomic transactions on objects, largely hiding the fact that the object store is distributed and replicated. This paper discusses the design of *ROSS*.

ROSS gives the programmer two types of object consistency. A strongly consistent transaction uses the conventional, quorum-based virtual partition algorithm [3]. To provide greater availability, however, the object store supports a weaker form of replica consistency in addition to the traditional strong consistency. *ROSS*'s notion of weak consistency has been designed to most efficiently support databases consisting of objects having limited interdependencies, although general database structures are possible. The nature of many applications is such that the probability of a conflict between transactions is low, perhaps because the frequency of updates is low or there is a one-to-one mapping between an object and its "administrator". Also, in many applications it is not necessary for read operations to always return the most recent version of an object. A distributed name service is an example of an application having these characteristics. Some types of applications, such as network monitoring, may wish to enforce consistency but permit transactions to occasion-

† This work was made possible by grants from the Natural Sciences and Engineering Research Council of Canada.

ally be "lost" in exchange for higher throughput. This is analogous to an application choosing to use unreliable datagrams to transmit real-time voice.

ROSS has been designed with the following goals in mind:

- High degrees of availability and scalability (e.g., via load balancing) are required. There is a copy of each object marked for replication in each replica. It is assumed that most of the time all replicas will be operational and so an "optimistic" approach is reasonable. When one or more replicas cannot be contacted, however, the object store should allow operations to proceed at the risk of rollback, when permissible. In this case, *ROSS* permits read and write operations to occur despite host or network failures.

- The object store must provide high performance. With weak consistency, this may be achieved by taking advantage of the characteristics of a common class of database applications where there is limited interdependency between objects and, therefore, a low probability that a transaction will be rolled back. A read operation need not return the most recent version of an object. While users may observe differences among the replicas, if all object stores become quiescent and each replica is operational during some subsequent synchronization operation, replicas should possess identical object stores.

- While certain efficiencies are possible if the replicas share a broadcast medium, only reliable point-to-point communication is required.

- The objects should be opaque to the object store, making the system application independent.

In the next section, the model of the object store is given and the two replica control algorithms are outlined. Objects and object types are presented in Section 3 followed by an overview of the functional components of the object store in Section 4. The interface to the object store is described in Section 5 and transactions in Section 6. Initialization and maintenance issues are addressed in Section 7. Section 8 concludes the paper.

2 Object Store Model

ROSS is a multi-threaded package [12] that can be integrated into an application process or configured as a server process running in its own address space. For efficiency, however, the object store thread is usually integrated into the server for a particular application (e.g., the Directory System Agent for the CCITT X.500 Directory[5]). An object store consists of application-specific objects and an *object store thread* that provides operations on the objects and is responsible for maintaining consistency. Within the object store, objects belong to a particular application and access is restricted to the application. There is no cooperation between object store threads in different processes.

Each replica usually resides on a different host and it is assumed that each pair of replicas can communicate reliably. Few assumptions are made concerning the objects that make up the database or the hosts on which the replicas reside. *ROSS* does not inspect the contents of an object, so any problems associated with heterogeneity must be handled by the client.

Figure 1 shows a possible configuration. Objects belonging to Application 1 are replicated on three hosts, while objects belonging to Application 2 are replicated on two hosts. The object store service is implemented as a separate server on Host `relay.cdnnet.ca`.

It is important for the programmer to understand the differences in semantics between strong and weak consistency so that the desired behaviour is obtained for a particular class of objects. In the case of strong consistency, one-copy serializability [8] is provided

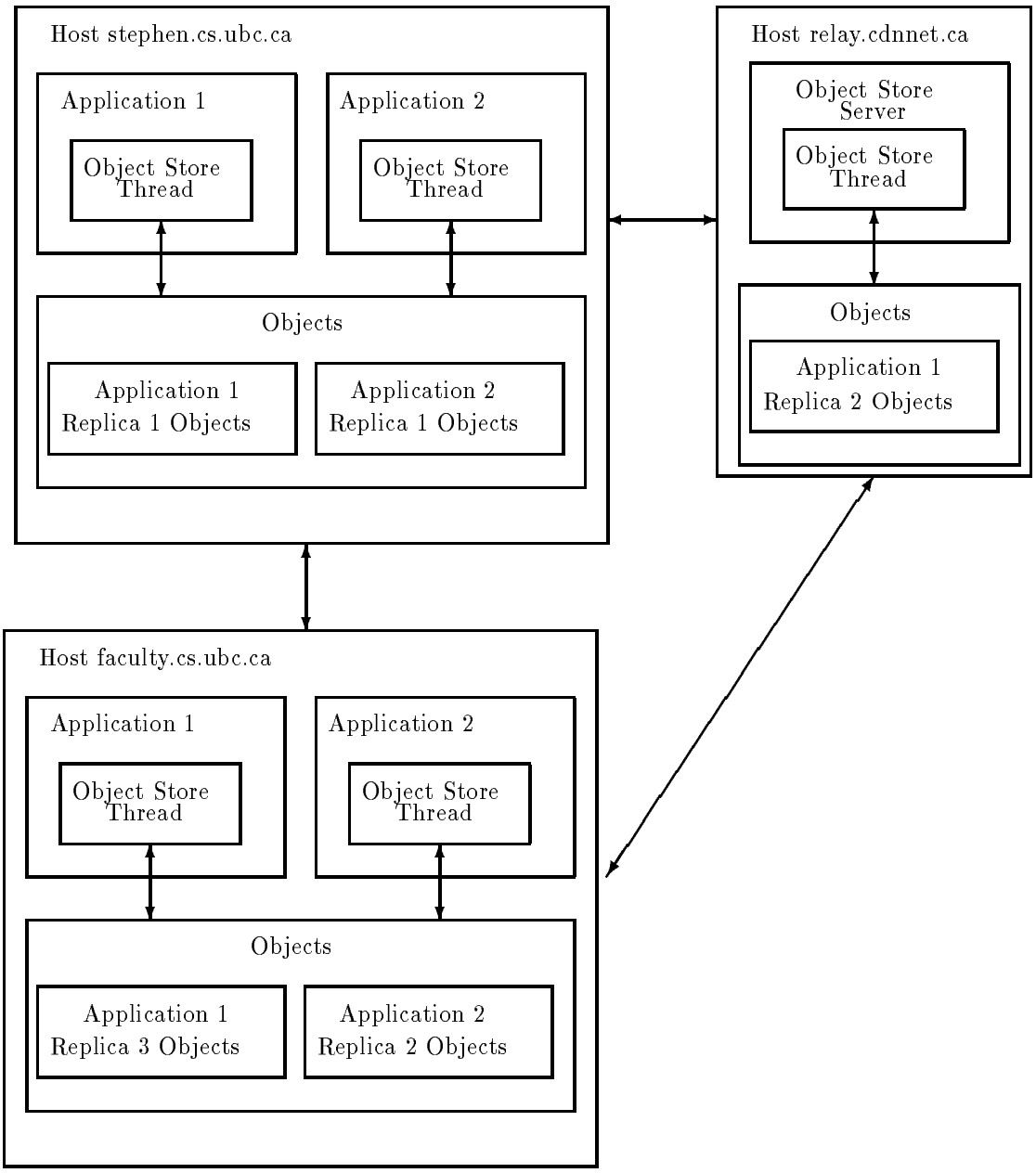


Figure 1: The Object Store Model

by *ROSS*.[2]

The virtual partition algorithm [3, 9] is used to implement strong consistency. The algorithm assigns each copy of an object a non-negative weight and defines a read threshold RT and write threshold WT such that both $2 * WT$ and $RT + WT$ are greater than the total weight of all copies of the object. A read quorum of an object is any set of copies with a weight of at least RT and a write quorum is any set of copies with a weight of at least WT.

An atomic transaction that is weakly consistent is committed in the usual way at the *originating replica* but may later be rolled back *asynchronously*. That is, although an indication is received that the transaction has been successfully committed locally, if it is found to conflict with a transaction executed at another replica it may be rolled back. A weakly consistent transaction may therefore be viewed as a nested transaction where the top-level commit operation is performed asynchronously or lazily. As a result, users may observe temporary inconsistencies.

The delay between when a transaction is locally committed and when a final decision is reached regarding global commitment depends on system parameters governing the frequency of update distribution and synchronization, as well as availability of replicas during the update and synchronization operations. A transaction cannot be globally committed until all replicas decide that it does not need to be aborted. It is possible for the programmer to determine the state of a transaction, including whether it has been "globally committed" (i.e., committed at all replicas) and is therefore immune to abortion.

Another characteristic of weakly consistent transactions is that the system enforces a *read dependency* on objects read during an *update transaction* (i.e., a transaction that modifies the object store). If a read dependency is expressed for an object that is not globally committed, the current transaction may be aborted if the object is rolled back to a different version than is named in the current transaction. This can result in a cascade of transactions that are aborted because a transaction near the beginning of a chain of dependencies must be rolled back.

The semantics of weak consistency are a unique characteristic of *ROSS*. Its goal is to provide the utmost availability in situations where conflicts among transactions are unlikely and temporary inconsistencies are acceptable. Many approaches to the problem of consistency in partitioned networks have been proposed [8]. A few that are relevant either to relaxing consistency constraints in favour of increasing availability or to asynchronous propagation of transactions are mentioned here. Allchin [1] discusses a suite of decentralized algorithms that do not achieve serial consistency but which provide high availability. Davidson's Optimistic Protocol [7] is similar in spirit to our approach but differs in the details of distributing updates and resolving inconsistencies. Boaz [2] has proposed an approach that adds fault tolerance to the two phase commit protocol by having the writer send updates to some write quorum and then continue; the remaining replicas are updated "in the background". The Camelot distributed transaction facility[6] provides lazy transactions that do not provide durability until a subsequent non-lazy update transaction commits.

Table 2 summarizes some of the differences between strong and weak consistency for an object store composed of ten replicas. The object store in this example is configured for maximum availability, with each replica having a copy of all objects and all objects having the same weight.

Note that in cases where strong consistency does not obtain the required quorum, the transaction must block or be aborted and re-executed. In the latter case, some additional mechanism is necessary to retry the transaction. A similar mechanism would be

[2] Users may observe different results from a read operation executed on different replicas simultaneously[9].

Transaction Type	# Of Replicas Needed To Update An Object	# Of Replicas Needed To Read An Object	Read Returns Same Object At All Replicas
Weak consistency	Can always write locally	Can always read locally	No
Strong consistency	10	Can always read locally	Yes
Strong consistency	6 (minimum write quorum)	5	Yes

Figure 2: Strong vs. Weak Consistency

used by weakly consistent transactions that are aborted by the system.

3 Objects, Types, and Properties

Each object has a variable-length, unique[3] object identifier (OID or `ObjOid`) associated with it. The OID consists of the name of the replica that created the object, the date and time of creation, and an application-specific string.

```
typedef short replicaID; /* See ObjType_Replica */
typedef int ObjTypeID;   /* Object identifier */

typedef struct AppID {
    char       *appString;
    int        appStringLength;
} AppID;

typedef struct ObjOid {
    replicaID  creationReplica;
    time_stamp creationTime;
    ObjTypeID  typeID;
    AppID      appID;
} ObjOid;
```

The object store supports different object *types*. Objects having the same `typeId` share the same data representation. The combination of this `typeId` and a set of immutable *properties* form a unique identifier for the object type within the object store. The properties determine access, storage, replication, recoverability, and consistency characteristics of all objects belonging to the type. Figure 3 shows the possible combinations of properties with some examples of applications.

Objects having the access property `ObjLocal` are strictly local to the replica on which they are created; `ObjRemote` objects can be accessed from any replica.

`ObjRemote` objects have replication property `ObjReplicated` or `ObjNonReplicated`. Property `ObjReplicated` marks an object as being fully replicated.[4] The property `ObjNonReplicated` indicates that there is a single copy of the object that can be accessed from other object stores.

Objects having the property `ObjVolatile` are not stored on secondary storage and will be lost if their replica crashes. Because these objects reside in the address space of the process that creates them, they are automatically destroyed when the process terminates. Objects that are `ObjPersistent` survive replica crashes.

Objects having the recoverability property `ObjAtomic` can be rolled back to their previous state in the event that the transaction with which they are associated aborts. Objects that are `ObjNonAtomic` cannot be rolled back and may be corrupted if a crash occurs during an update. A transaction cannot mix operations on objects having these two properties.

Finally, replicated objects that are `ObjAtomic` can be either strongly or weakly

[3] In this paper, all unique identifiers are unique network-wide across time.

[4] A *fully replicated object* is stored at each replica, while a *partially replicated object* is stored at two or more replicas.

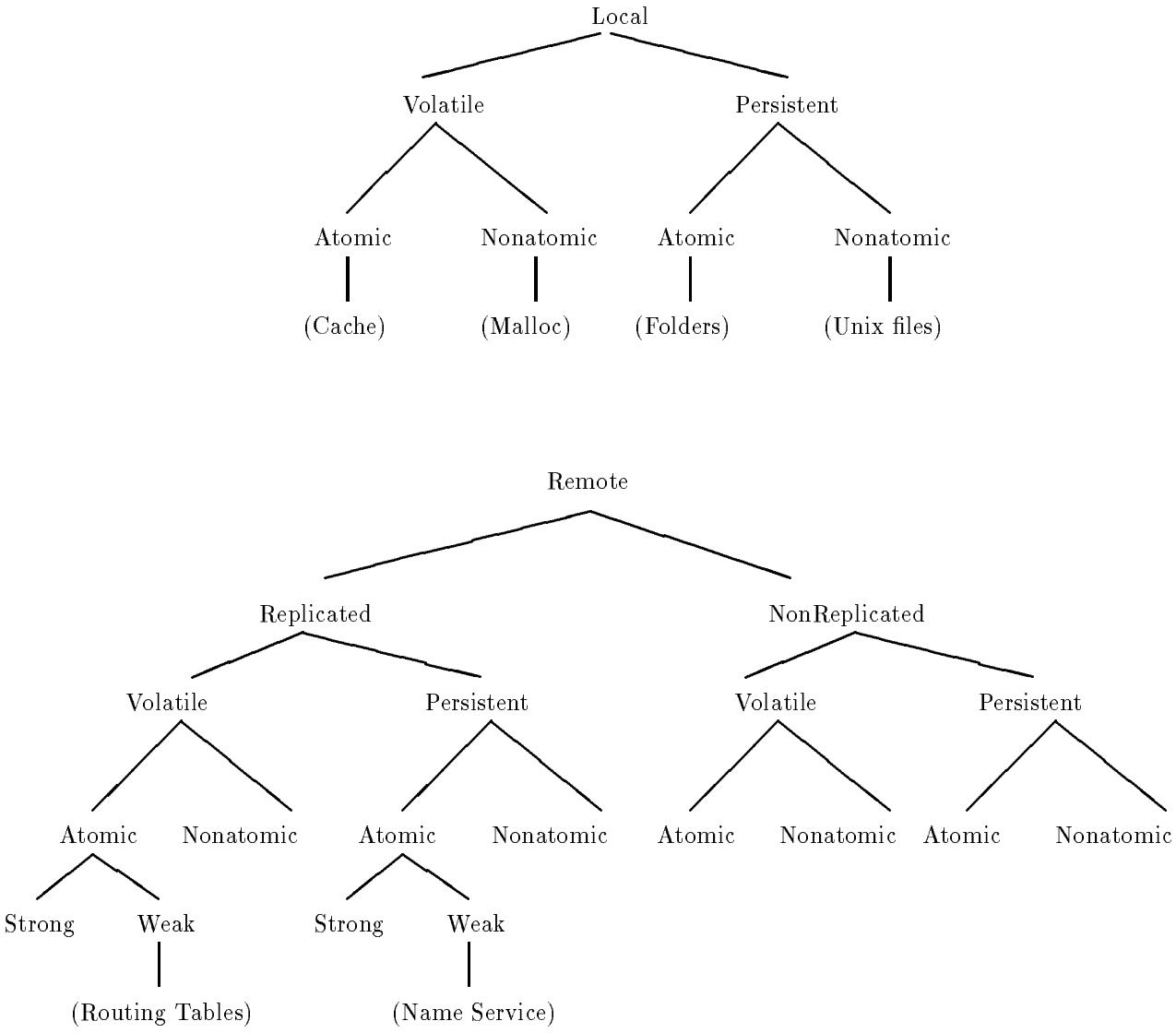


Figure 3: Object Properties

consistent. Objects that have `ObjConsStrong` consistency are guaranteed to always be identical at all operational replicas. Objects that are `ObjRemote` and `ObjNonReplicated` must have `ObjConsStrong` consistency and objects that are `ObjLocal` and `ObjAtomic` are always `ObjConsStrong`. Objects having `ObjConsWeak` consistency have weak consistency semantics. A transaction cannot involve operations on types of both consistency modes. A transaction is said to be of type `ObjConsWeak` if all objects referenced in the transaction are of that type, otherwise the transaction is of type `ObjConsStrong`.

```
enum ObjPropAccess
    { ObjLocal, ObjRemote };
enum ObjPropStorage
    { ObjVolatile, ObjPersistent };
enum ObjPropReplication
    { ObjReplicated, ObjNonReplicated };
enum ObjPropRecoverability
    { ObjAtomic, ObjNonAtomic };
enum ObjPropConsistency
    { ObjConsStrong, ObjConsWeak };
```

The system itself uses special *system objects* of various types. A system object has a well-known (reserved) OID[5]. A system object called the *type directory* contains information about all known types. It contains an entry for itself (i.e., it is the only instance of object type `TypeDirectory`), and has properties `ObjReplicated` and `ObjConsStrong`. System objects of type `ObjType_PropInfo` contain the properties for a particular object type.

```
struct ObjType_PropInfo {
    enum ObjPropAccess         access;
    enum ObjPropReplication    replication;
    enum ObjPropConsistency    consistency;
    enum ObjPropStorage        storage;
    enum ObjPropRecoverability recoverability;
} ObjType_PropInfo;

struct ObjType_TypeEntry {
    ObjTypeID typeID;
    char      *typeName;
    ObjOid    typeInfo;
};
Set Of (struct ObjType_TypeEntry)
    ObjType_TypeDirectory;
```

[5] There could be a directory of system objects to map external, conventional names to OIDs. In this case only the OID of the directory need be well-known.

To illustrate this, four type-related objects that might appear in an object store are shown in Figure 4. The type directory contains type information for three types: itself, type `Type`, and type `X.500`. For each of these object types there is an object (of type `Type`) containing the access, consistency, storage, and recoverability information for the type.

For each object type, routines are specified at the time the database is opened to encode and decode the object and to generate keys for the object. The encode routine is typically used to flatten a linked list of data structures representing the object into a contiguous buffer; the decode routine restores the data structure. In some applications, a general purpose encoding scheme such as ASN.1 might be chosen to provide a certain degree of machine and programming language independence. The index routine is used to generate a set of keys for a given object. The keys are used in an inverted index, allowing efficient retrieval of the object. Because the client may not reside in the same address space (or even on the same host) as the object store being accessed by an operation, these functions are executed by the client.

4 Functional Overview

The functional components of the object store can be divided into four parts: *nested atomic transactions*, *update distribution*, *recovery*, and *synchronization*. These components are discussed in detail in later sections.

The *nested atomic transaction* component presents the programmer with a familiar transaction interface called the *object layer*. The object layer provides basic I/O operations on objects, nested atomic transactions, and cache management functions. These functions will be described in the next section.

The update distribution component of the object store is invoked when a top-level commit occurs on a transaction involving one or more `ObjRemote` objects. When consis-

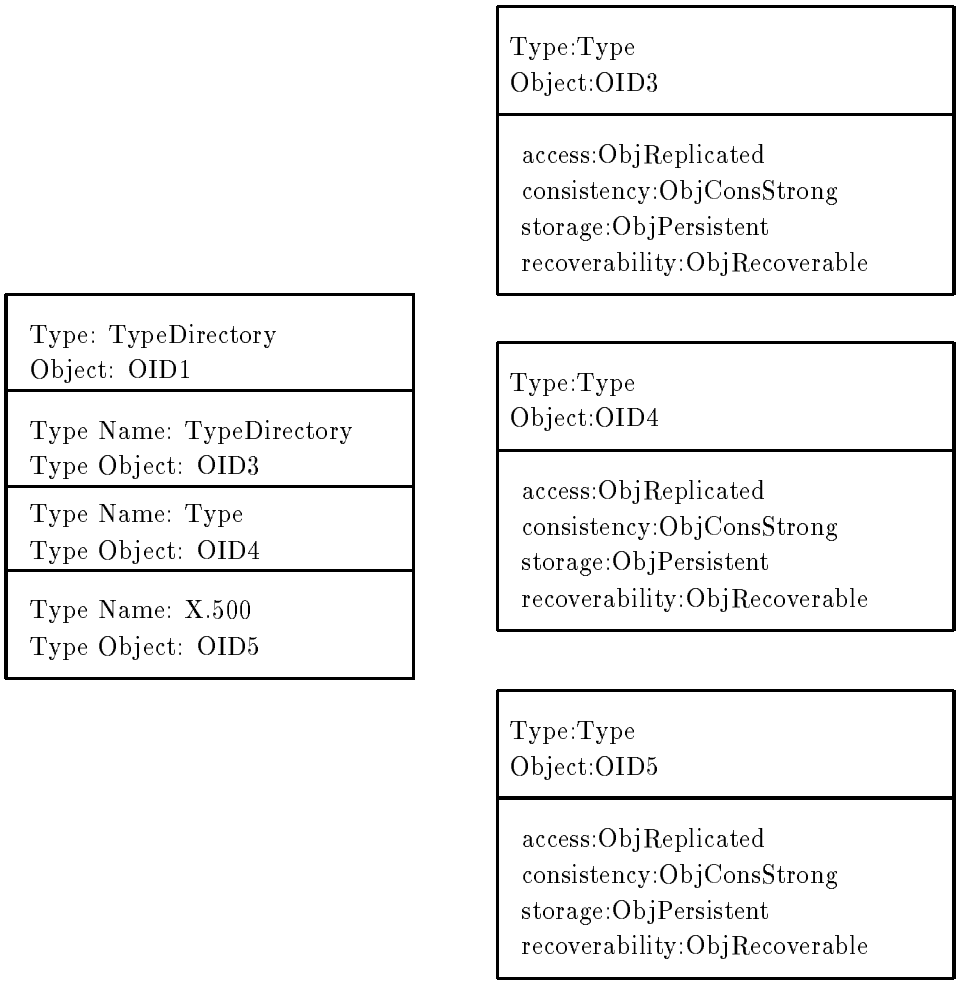


Figure 4: Object Type Information

tency contraints are relaxed in the case of `ObjConsWeak` transactions, update distribution attempts to keep as many copies as possible "current" (the *write-all-available* approach [3]). Assuming that most of the time all replicas receive and process a transaction, the replicas will be mutually consistent. If one or more replicas cannot be reached or do not successfully complete the transaction, the synchronization component can be used to achieve mutual consistency later. Update distribution may result in the aborting of transactions.

When a replica is restarted it performs recovery operations. In addition to cleaning up after partially completed local transactions, the replica attempts to contact other replicas to exchange updates. This helps to limit distribution of old transactions and reduce the number of rollbacks required later.

The synchronization component is run periodically to try to complete (i.e., abort or globally commit) transactions not fully distributed by update distribution operations and not obtained during replica recovery after a crash. These transactions are the result of a network partitioning. It is not necessary for all replicas to be available at the start of synchronization, although all replicas taking part must be operational for its duration. Loosely speaking, as long as all replicas eventually participate in synchronization, transactions will be completed.

Update distribution, recovery, and synchronization are described in greater detail in Sections 6.1, 6.3, and 6.4, respectively.

5 The Object Layer

The object layer sits between the application and a database; `tdbm` [4] is currently used. It provides hash-based access to database items within the context of nested atomic transactions. `Tdbm` uses strict two phase locking[11] and provides support for the two phase commit protocol.

The interface to the object layer is described in this section. Most object layer routines return a result code (`ObjRc`).

5.1 Type Initialization and Termination

Before operations can be performed on objects of a given type, the database containing the objects must be named and routines may be specified to encode, decode, and generate keys for the objects. Because the client may not reside in the same address space (or even on the same host) as the object store being accessed by an operation, these functions are executed by the client.

The encoding routine is typically used to flatten a linked list of data structures representing the object into a contiguous buffer; the decoding routine restores the data structure. In some applications, a general purpose data representation method such as ASN.1 might be chosen to provide a certain degree of machine and programming language independence. In the case of ASN.1, a compiler can automatically generate encoders and decoders. In many cases the encoded representation requires less space. The CASN1 ASN.1 compiler [12], which uses the Basic Encoding Rules, is currently used by *ROSS*.

An indexing routine may be specified to generate a set of keys for a given object. These keys are returned as a list structure. The keys are used to construct inverted indexes, allowing efficient retrieval (using `ObjLookup()`) of objects containing the key. As an example of how this might be used, suppose objects of a given type are structured as attribute-value pairs. If "Colour" is an indexed attribute and if some objects have a value of "Blue" for this attribute, then the indexing routine might generate a key such as "Colour=Blue" from this attribute-value pair. The system will automatically maintain a system object that consists of the OIDs of all objects having this key. The precise structure of the key is up to the indexing routine. Indexes for remote, non-replicated

objects are stored at the same replica as the objects themselves.

```
typedef void *ObjInst;

typedef struct {
  void (*encode)(ObjInst object,
                 ObjInst *encoded, long *length);
  void (*decode)(ObjInst *decoded, ObjInst object,
                 long length);
  Set Of(OCTS) (*index)(ObjInst object);
} ObjTypeHandlers;

typedef struct ObjConfig {
  int mode;
  int pagesize;
  int allocunits;
} ObjConfig;

ObjRc
ObjInit(char *pathname, ObjConfig *config,
  ObjType *objType, ObjTypeHandlers *handlers)
```

An object type is initialized using `ObjInit`; it will be called once for each user-defined object type that the client will reference. The `pathname` names the database containing the objects; it may be `NULL` if both the database and the object type are volatile. Configuration parameters `config` are passed on to the underlying database. Objects of the given `objType` are associated with the database. A particular `objType` can only be associated with a single database; parameters of a subsequent call override the previous ones.

The `encode` and `index` functions are implicitly called when an object is written. The `decode` function is implicitly called when an object is read.

When objects of a particular type will no longer be referenced, `ObjTerm()` can be called to free resources. There must not be any active transactions on objects of this type. If the database is volatile, it will disappear if there are no longer any references to it.

```
ObjRc
ObjTerm(ObjType *objtype)
```

5.2 Nested Transactions

A transaction is initiated using `ObjBegin()` and terminated by `ObjCommit()`, `ObjAbort()`, or implicitly if the client exits. All operations on objects occur within the context of a transaction. Any transaction may create one or more subactions (also called subordinate transactions or children). Different transactions may execute concurrently. Note that the unit of concurrency is the transaction; multiple threads of control should not concurrently execute operations as part of the same transaction. In the current implementation, a transaction may not involve objects residing in different databases; a transaction is implicitly associated with a database based on the first I/O operation it performs.

`ObjBegin()` begins a new transaction and returns a unique transaction identifier. If `parent` is not `NULL`, the new transaction will be a subaction, otherwise it will be a topaction (top-level transaction).

```
ObjRc
ObjBegin(ObjTid *parent, ObjTid **child)
```

`ObjCommit()` attempts to commit the transaction. A transaction cannot commit while it has any subactions. If the return code is `OBJ_VALIDATION_FAILED` then the topaction was aborted and should be retried.

```
ObjRc
ObjCommit(ObjTid *tid)
```

A transaction is aborted using `ObjAbort()`. If a program terminates while a transaction is in progress, `ObjAbort()` is implicitly performed by the object store. A transaction cannot abort while it has any child transactions.

```
ObjRc
ObjAbort(ObjTid *tid)
```

5.3 Object I/O

A new type is created by passing a unique object type identifier (`objType`) to `ObjNewType()`.

```
ObjRc
ObjNewType(ObjTid *tid, ObjType *objType)
```

A new object is created using `ObjNew()` and a globally unique object identifier (`ObjOid`) is assigned to it. The `ObjOid` is returned.

```
ObjRc
ObjNew(ObjTid *tid, ObjType *objType, ObjOid **oid)
```

An object is destroyed using `ObjDispose()`.

```
ObjRc
ObjDispose(ObjTid *tid, ObjOid *oid)
```

An object is read using `ObjMap()`. The `object` parameter is set to point to the object.

```
ObjRc
ObjMap(ObjTid *tid, ObjOid *oid,
       ObjInst *object)
```

`ObjSecure()` is used to write an object. The object must have previously been read or created by this action. Any pointers to the previous instance of the object that this action may have will be invalid.

```
ObjRc
ObjSecure(ObjTid *tid, ObjOid *oid,
          ObjInst object)
```

The set of objects of a particular type that are indexed by a key can be determined using `ObjLookup()`). The `OCTS *` argument is a structure consisting of a key string and a length. A list of OIDs indexed by the key is returned.

```
ObjRc
ObjLookup(ObjTid *tid, ObjType *objType,
          OCTS *key, Set Of(ObjOid) *oidSet)
```

An object type can have an `ObjOid`, called the root `ObjOid`, associated with it. An application program can set (and reset) the root `ObjOid` using `ObjSetRoot()` and get it using `ObjGetRoot()`. This provides a simple way for an application to avoid having to deal with an initial `ObjOid` by some external means. Presumably the root `ObjOid` is some kind of `ObjOid` directory or the root of a tree structure.

```
ObjRc
ObjSetRoot(ObjTid *tid, ObjOid *rootOid)

ObjRc
ObjGetRoot(ObjTid *tid, ObjType *objType,
        ObjOid **rootOid)
```

5.4 Miscellany

After a transaction has read an object using `ObjMap`, it may elect to release the lock held on the object by calling `ObjRelease`. The transaction's copy of the object is freed and the object is not considered during commit-time validation. This function should be used with care if database consistency is to be retained. The function might be used, for example, by a tree locking protocol.

```
ObjRc
ObjRelease(ObjTid *tid, ObjOid *oid)
```

The following functions return a pointer to a (static) text buffer describing the given error and a classification of an error code, respectively.

```
char *
ObjErrorString(ObjRc rc)

enum {
    OBJ_NOERROR, OBJ_WARNING, OBJ_INTERNAL,
    OBJ_FATAL, OBJ_BADNEWS, OBJ_CORRUPTED
} ObjErrorClass;

ObjErrorClass
ObjRcClass(ObjRc rc)
```

6 Transaction Processing

Components of transaction processing are described in this section.

6.1 Update Distribution

The update distribution component attempts to execute a transaction on all replicas at the time a transaction is committed. It is responsible for both strong and weak consistency transactions. The locally executed portion of all transactions is based on an optimistic, single-site nested transaction model[10].

6.1.1 Strong Consistency Transactions

If all objects involved in the transaction have consistency `ObjConsStrong`, the transaction enforces strong consistency and the client blocks until the transaction completes. The virtual partition algorithm requires a write quorum to be obtained before the transaction

can be committed using the two phase commit protocol.

6.1.2 Weak Consistency Transactions

If all objects involved in the transaction have consistency `ObjConsWeak`, the transaction permits weak consistency and a write-all-available approach is used to update as many replicas as possible. Note that a two phase commit protocol [3] is *not* used when updating the available replicas, as is commonly done. If some replicas receive updates before it is determined that the updates must be rolled back, update distribution is terminated and a special UNDO operation is sent to the replicas that received the updates. Ideally, all replicas are operational and the transaction can be completed. The client blocks only while the transaction is committed locally. A system parameter determines how frequently update distribution is performed.

Because update distribution may not contact all replicas, inconsistencies could arise if they are not prevented. If a replica misses update distribution of a transaction that creates an object, a later update distribution may refer to an object that is not known to the replica. If a replica misses update distribution of a transaction that modifies an object, a read dependency may be violated during a subsequent update distribution.

An optimistic solution to this approach allows temporary inconsistencies to exist and lets the synchronization resolve any problems.

Alternatively, a pessimistic solution that enforces stricter consistency may be used. By associating version identifiers with objects, a comparison of the object read by the transaction to that of the replica's copy of the object can identify an inconsistency and cause the transaction to be ignored. This is in keeping with the notion of update distribution as only a "best effort" attempt to keep replicas up-to-date. The synchronization operation (Section 6.4) is also used here to reconcile differences among the replicas. Either solution can be extended to attempt to locate and execute the missing updates.

6.2 Transaction Logging

Transaction logging is an implicit part of every update transaction, carried out as a transaction is built and completed as part of commit processing. If a logging operation fails, the transaction is aborted.

When update distribution fails to complete a transaction that is weakly consistent, the transaction must be logged so that it can be rolled back if necessary by synchronization. Different information is logged by strongly consistent transactions for the "View Update transaction" [3, p. 304], a transaction issued by the system itself. This special synchronization operation uses this log to efficiently bring up-to-date all objects for which there is a read quorum when a new view is created.

Several new system types are used to implement logging. Objects of these types are `ObjLocal` and `ObjConsStrong`. Each update transaction results in a system object, called a *log entry*. The transaction identifier (`ObjTransOid`) returned by `ObjBegin()` and recorded in the log directory is actually the OID of the log entry.

The *log directory* is a system object of type `LogDirectory`. It contains directories for both `ObjConsWeak` and `ObjConsStrong` transactions. The logs are structured according to the virtual partition algorithm's notion of a *view*; i.e., those replicas with which the originator of a transaction can communicate at commit time. Associated with each view is a unique *view identifier* (VIEWID).

```
typedef struct VIEWID {
    replicaID  creationReplica;
    time_stamp creationTime;
} VIEWID;

typedef struct ObjType_View_Logs {
    VIEWID              viewID;
    Set Of(ObjTransOID) transOIDs;      /* log objects */
} ObjType_View_Logs;

struct ObjType_LogDirectory {
    Set Of(ObjType_View_Logs) consStrongLogs;
```

```
    Set Of(ObjType_View_Logs) consWeakLogs;
} ObjType_LogDirectory;
```

The structure of these logs is detailed in the next two sections.

6.2.1 Logging of Strong Consistency Transactions

Most strongly consistent update transactions create a log entry that is an instance of type `LogEntryConsStrong`. If the update distribution involves all replicas, the transaction need not be logged.

For each view that has a write quorum, an object is created that records the OIDs of objects involved in update transactions. All replicas that have seen a view, either by being a member of the view's write quorum or through a view update transaction, will have the same log information for the view. A view's log is destroyed when all replicas have seen it. Although a `LogEntryConsStrong` log entry is technically `ObjLocal`, it is replicated as part of the virtual partition algorithm.

A log entry containing the following information is created when a new view is created and updated when each `ObjConsStrong` transaction within the view is committed:

- The set of objects that have been created or otherwise modified as part of a transaction (duplicates may be removed).
- The set of replicas that have seen this view.

```
struct ObjType_LogEntry_ConsStrong {
    Set Of(ObjOid)    modifiedObjects;
    Set Of(replicaID) seenbyReplicas;
} ObjType_LogEntry_ConsStrong;
```

If a read quorum is found to exist during the view update transaction, a replica can quickly determine which, if any, of its objects are out-of-date and where the most up-to-date copies of the objects can be found.

6.2.2 Logging of Weak Consistency Transactions

A log entry that is an instance of type `LogEntryConsWeak` is created at the originating replica by an update transaction of type `ObjConsWeak`. Although these log entries are `ObjLocal`, they are transmitted as part of update distribution, recovery, and synchronization.

The following information is logged when an `ObjConsWeak` transaction is locally committed:

- The time and date (used to resolved conflicts).
- A list of operations performed within the transaction and the OID created or referenced by each operation (`ObjNew()`, `ObjDispose()`, `ObjMap()`, or `ObjSecure()`).
- The current status of the transaction.
- The set of replicas that have seen and applied the transaction.
- A copy of the previous version of the object (more precisely, the object's instance data) in case rollback is required.
- If some measure of the computational resources used by the transaction is required for conflict resolution, this data will also be logged.

```
/* Instance variable (pointer to instance data) */
typedef void *ObjIV;
typedef struct LogEntryOperation {
    enum op { OP_NEW, OP_DISPOSE,
              OP_MAP, OP_SECURE } op;
    /* Object created or referenced */
    ObjOid op_object;
    ObjIV previous_instance_value;
} LogEntryOperation;

struct ObjType_LogEntry_ConsWeak {
    time_t                        commit_date;
    Set Of(ObjLogEntryOperation) operations;
    enum trans_state {
        TRANS_COMMITTING, TRANS_ABORTING,
        TRANS_COMMITTED
    } trans_state;
```

```
    Set Of(replicaID)           seenbyReplicas;
} ObjType_LogEntry_ConsWeak;
```

As in the logging of strong consistency transactions, a replica associates each log entry with a VIEWID. This increases the efficiency of update distribution, recovery, and synchronization since missing groups of updates can be located easily.

As transactions are completed they can be removed from the transaction log. A transaction is globally committed if all replicas have seen the transaction and none have aborted it.

6.3 Recovery

Every time a replica is started it must clean up after any transactions that were in progress when it went down. The `tdbm` database assists by providing a list of transactions that have not completed the two phase commit protocol. During this recovery period, the replica must also contact other replicas to obtain update distributions it may not have seen.

6.4 Synchronization

Synchronization is a multi-phase operation that resolves conflicts among weakly consistent transactions.

6.4.1 Phase 1

Phase 1 involves forming a virtual ring of as many operational replicas as possible. Replicas that are unreachable are excluded from a virtual ring. In the event of network partitioning, concurrent synchronization operations can occur; although to keep the system simpler a quorum could be required.

As the ring is constructed, a replica forwards its transaction log and the transaction logs of other replicas it has received during this phase. Once the virtual ring is established, each replica has complete log information on all locally committed transactions among the participating replicas. The operation aborts if a ring can't be formed or a replica crashes.

6.4.2 Phase 2

Phase 2 involves examining the transactions to determine if there are conflicts and resolving them by aborting one or more of the transactions involved. Conflict resolution is the topic of Section 6.5.

6.4.3 Phase 3

In Phase 3, the final phase, individual replicas roll back locally committed transactions as determined by the conflict resolution algorithm and globally commit transactions.

6.5 Conflict Resolution

During update distribution, recovery, or synchronization, a replica may determine that a pair of transactions that have been committed at different replicas have conflicting accesses to the same logical object. To attain one-copy serializability, these conflicts must be resolved.

A transaction that has not been locally committed is aborted if it interferes with update distribution or synchronization.

Note that the relative order in which weakly consistent transactions are locally committed should be preserved during update distribution and synchronization (e.g., because a transaction performing an `ObjNew()` should be handled before a transaction that references the object and because read dependencies may cause a later transaction to be aborted). If update distribution of a transaction that creates an object incompletely updates replicas, then subsequent transactions on the object must be queued behind the transaction creating the object for those replicas that have not created the object. Certain "anomalous" operations (e.g., use of an OID that is not known to a replica because the object has not been created or it has been destroyed) can be prevented in the

case of weakly consistent objects if this ordering is maintained.

There are two sources of conflict that can arise when trying to globally commit a transaction:

1. Two transactions update the same logical object.
2. The read dependencies of a transaction are violated.

A user-specified function (or program in the case of a generic object store server) can be provided to impose some total ordering on transactions for the purpose of resolving conflicts. This permits a finer degree of granularity in resolving a conflict, including determining whether a conflict really exists. Unlike the object store, the user program knows the internal structure of an object. It is conceivable that in some instances the user's function could determine the differences between two versions of the same object, perhaps creating a new version reflecting the two sets of changes. An example of this is the update of two or more independent attribute values of an object in two or more partitions. If the user has not specified a resolver, the default behaviour is to rule in favour of the "newest" transaction.

Conflicts involving read dependencies can be resolved based on rolling back the transaction leading to the shortest cascade or the cascade that appears to have taken the shortest time to perform originally.

Each replica constructs a *conflict resolution table* (Figure 5) from the collective logs. Each transaction is assigned a row in the table, and each object that is referenced by a transaction has a column. An entry in the table records the operation (`ObjMap`/`ObjLookup`/`ObjGetRoot`, `ObjSecure`/`ObjSetRoot`, `ObjDispose`) performed on a particular object by a particular transaction. `ObjNew` operations are not entered since they cannot cause a syntactic conflict. Because a transaction may perform different operations on the same object, a precedence is assigned to the operations and a higher priority operation supersedes one of lower priority in the table. The priorities of operations, from lowest to highest, are: `ObjMap`, `ObjSecure`, and `ObjDispose`. Note that `ObjSecure` effectively implies `ObjMap` since an object must be read before it can be written. If a transaction performs each of these operations on a particular object, for example, only the `ObjDispose` is recorded in the table.

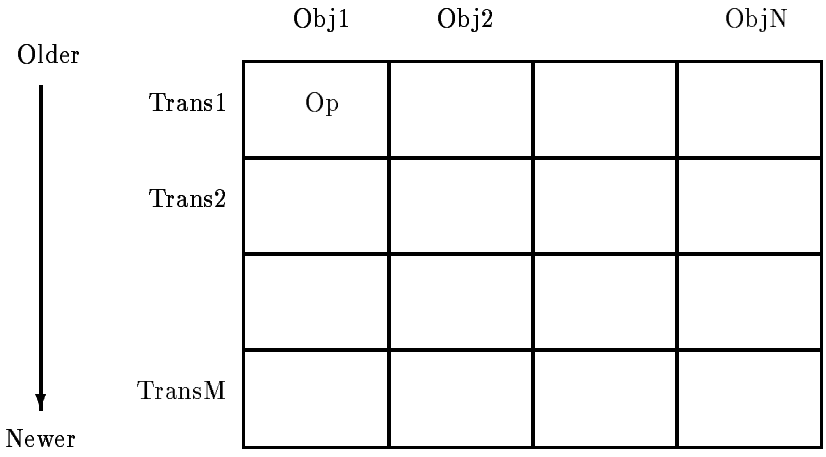
A transaction is involved in a conflict if, scanning down each column in the table belonging to an object referenced by the transaction:

- The operations are not all `ObjMap` operations or
- The operations are not all `ObjDispose` operations or
- There is more than one entry, one of which is an `ObjSecure` operation

Transactions that were committed at the same replica cannot conflict.

If the user has not provided a conflict resolution method, a simple date-based algorithm is used as the default. With one exception, the most recent transaction succeeds and all older conflicting transactions are aborted. This was chosen because it is simple, favours the most up-to-date information, and may tend to reduce the number of rollbacks (e.g., a replica crashes and stays down for an extended period while holding transactions that have not been distributed). The exceptional case is that an `ObjDispose()` operation takes precedence over all other operations, independently of the transaction date. This is because the intent behind destroying an object is to destroy it everywhere and operations should not be permitted on a destroyed object. Transactions that destroy the same object do not conflict. When a transaction is aborted its entries are removed from the table.

Figure 6 is a conflict resolution table consisting of four transactions (performed at different replicas) and four objects. The default algo-



Op = ObjDispose, ObjMap, or ObjSecure

Figure 5: Format of the Conflict Resolution Table

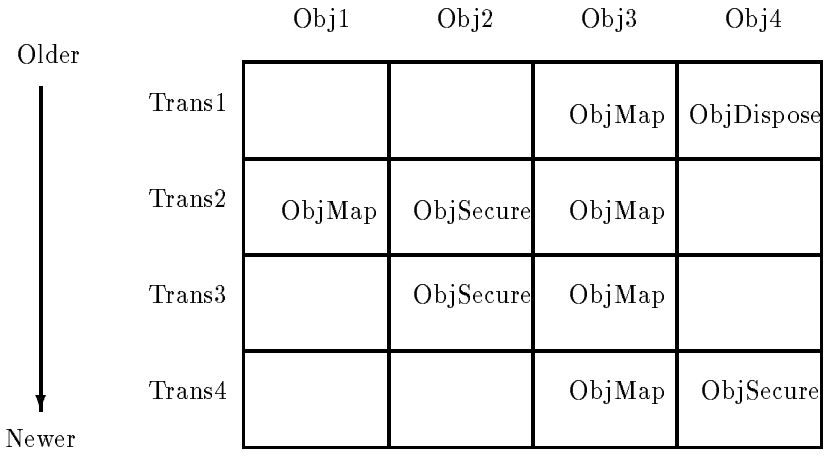


Figure 6: A Conflict Resolution Table

rithm will abort **Trans2** (because of **Trans3**) and **Trans4** (because of **Trans1**).

Figure 7 is a conflict resolution table that results in a cascade of rollbacks. The first three transactions were locally committed at the same replica while the other transactions were committed at different replicas. The default algorithm will abort **Trans1** (because of **Trans2**, **Trans4**, and **Trans5**), **Trans2** (because **Trans1** was aborted and **Trans6**), **Trans3** (because **Trans2** was aborted), and **Trans6** (because of **Trans4** and **Trans5**).

7 Initialization and Maintenance

The system is initialized by a program that creates the necessary system objects. A system maintenance program is used to examine and modify system objects. The maintenance program lets an administrator:

- display and alter system objects, including logs
- print statistics, including information on update distributions and synchronization operations
- remove objects
- disable and enable a replica
- start the synchronization operation

When a database is first created, a system object of type **Replica** called the **replica object** is created to contain information about the database itself, including the names of all replicas and user-specified parameters. It has consistency **ObjConsStrong** and access **ObjReplicated**. With this approach, no special mechanism is required to distribute operational information to the replicas. The order of replicas within the list determines the order in which the virtual ring is formed.

```
typedef struct ReplicaEntry {
    replicaID replica;
    enum replica_state {
        REPLICA_JOINING, REPLICA_LEAVING,
        REPLICA_INACTIVE, REPLICA_ACTIVE
    } replica_state;
    char   *replica_access_point;
        /* eg., address + port number */
};

struct ObjType_Replica {
    int     version;
    time_t  lastSync;
    int     syncFrequency;
    Set Of(ReplicaEntry) replicaRing;
} ObjType_Replica;
```

Each replica also maintains a non-replicated system object.

```
/* For separate server */
typedef struct ResolverProgram {
    ObjTypeID typeID;
    char *resolver;  /* Program to invoke */
} ResolverProgram;
typedef struct ObjType_LocalInfo {
    int                     updateFrequency;
    time_t                  lastUpdate;
    Set Of(ResolverProgram) resolverProgs;
} ObjType_LocalInfo;
```

The location of the database files for a particular replica are provided as arguments to the object store service when it is started.

It is possible to parameterize the system to adjust trade-offs between the time taken to achieve global commitment, service availability, computational and communication expense, and the probability of abortion.

Adding a replica to an object store involves adding its name to the list of replicas in the replica object. The object store must be initialized at the new replica (with the new replica object) before the other replicas are informed by a normal update of the replica object. When the replica joins a view containing a read quorum for the first time, it can request a complete downloading of **ObjReplicated** objects. Alternatively, an offline method of obtaining these objects could be provided. Joining the object store is a two-phase operation: **REPLICA_JOINING**, followed by **REPLICA_ACTIVE**.

Removing a replica from the list requires that all **ObjConsWeak** transactions originating at the replica be completed. Leaving the

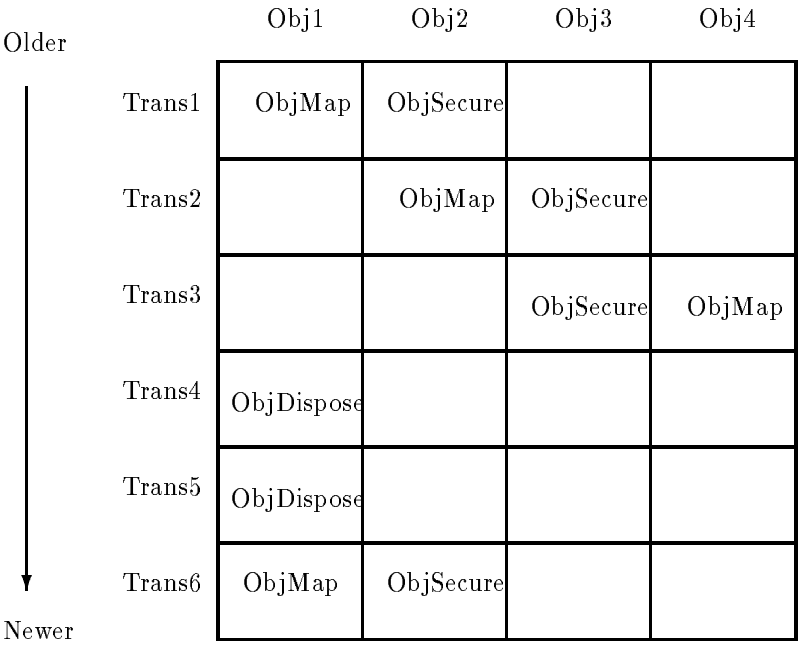


Figure 7: A Conflict Resolution Table Resulting in Cascading Rollbacks

object store is also a two-phase operation: `REPLICA_LEAVING`, removing the replica from the replica object, and `REPLICA_INACTIVE`. A replica marked as being `REPLICA_INACTIVE` will not perform update transactions or act as a member of a quorum.

8 Conclusions

The design of an object store featuring both strong and weak consistency has been presented. This allows *ROSS* to be used in those situations that call for one-copy serializability as well as those where availability and performance are paramount.

A non-distributed version of *ROSS* is currently being used by the EAN X.500 name service and implementation of *ROSS*'s distributed features is underway. Future work includes an evaluation of our approach to weak consistency in the context of X.500 and other distributed applications.

References

[1] J. E. Allchin. "A Suite of Robust Algorithms for Maintaining Replicated Data Using Weak Consistency Conditions", *Proc. Third IEEE Symp. on Reliability in Distributed Software and Database Systems*, Oct. 1983, pp. 47-56.

[2] Boaz Ben-Zvi. "Disconnected Actions: An Asynchronous Extension to a Nested Atomic Action System", MIT/LCS/TR-475, Jan. 1990.

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

[4] Barry Brachman and Gerald Neufeld. "TDBM: A DBM Library With Atomic Transactions", *Proc. of the USENIX Summer Technical Conference*, June 1992, pp. 63-80.

[5] CCITT. "Unofficial 'final' version of the X.500 series: The Directory", Dec. 1988.

[6] Joshua Bloch. "Camelot and Avalon: A Distributed Transaction Facility", edited by J. Eppinger, L. Mummert, and A. Spector, Morgan Kaufmann, 1991, pp. 21-56.

[7] Susan B. Davidson. "Optimism and Consistency in Partitioned Distributed Database Systems", *ACM Trans. on Database Systems*, vol. 9, no. 3, (Sept. 1984), pp. 456-481.

[8] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. "Consistency in Partitioned Networks", *Computing Surveys*, vol. 17, no. 3, (Sept. 1985), pp. 341-370.

[9] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. "An Efficient, Fault-Tolerant Protocol for Replicated Data Management", *Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, March 1985, pp. 215-228.

[10] Robert Gruber. "Optimistic Concurrency Control for Nested Distributed Transactions", MIT/LCS/TR-453, June 1989.

[11] J. Elliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

[12] Gerald Neufeld, Murray Goldberg, and Barry Brachman. "The UBC OSI Distributed Application Programming Environment – User Manual", Technical Report 90-37, Department of Computer Science, University of British Columbia, Jan. 1991.