Distributed Object Management in Thor

Barbara Liskov

Mark Day

Liuba Shrira

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge MA 02139
USA

{liskov, mday, liuba}@lcs.mit.edu

Abstract

Thor is a new object-oriented database management system (OODBMS), intended to be used in heterogeneous distributed systems to allow programs written in different programming languages to share objects in a convenient manner. Thor objects are persistent in spite of failures, are highly likely to be accessible whenever they are needed, and can be structured to reflect the kinds of information of interest to users. Thor combines the advantages of the object-oriented approach with those of database systems: users can store and manipulate objects that capture the semantics of their applications, and can also access objects via queries.

Thor is an ongoing project, and this paper is a snapshot: we describe our first design and a partial implementation of that design. This design is primarily concerned with issues related to the implementation of an OODBMS as a distributed system.

1 INTRODUCTION

Distributed systems contain different kinds of computers, and users write programs in different programming languages. The programs have persistent data that they need to share. This paper describes a new system called Thor that is intended to provide a convenient sharing mechanism.

Thor allows users to store and manipulate objects that capture the semantics of their applications. Users can access these objects by high level path names, by navigation, and also by means of queries.

Thor provides users with a universe of persistent objects. Objects in the universe can refer to one another, so that useful object structures, such as graphs and trees, can be constructed. All accesses to Thor objects occur within atomic transactions so that objects can remain consistent in spite of concurrency and failures.

To support heterogeneity Thor provides a language-independent type system that allows programs written in different programming languages to share data. The type system is hierarchical to support program evolution. Thor objects are abstract and encapsulated: each object has operations (or methods) that can be used to interact with it. Users can extend the type system by defining new, abstract types. A type is defined by a specification; specifications are independent both of any programming language used to access the type's objects and of the language used to implement the new type.

Thor provides a persistent root and high-level path names that can be used to name objects relative to the root. The root provides the basis for persistence: All objects accessible (directly or indirectly) from the root persist. If a persistent object ceases to be accessible, it will be garbage collected. Persistent objects are stored reliably, so that with high probability they will not be lost in the case of failures, and they are highly-available, so that they are highly likely to be accessible when needed. High availability and reliability are supported by a novel implementation strategy based on replication. In addition, Thor supports secure sharing by allowing users to define access control for objects.

Thor is a distributed system in which objects are stored at server nodes that are distinct from the machines where client programs reside. Nevertheless, it will provide clients with fast access to objects. Thor front ends run at client machines to cache and prefetch objects and to run operations on objects locally, thus reducing delay as observed by the clients. This architecture provides high performance and also scalability, since front ends offload work from the servers and extra servers can be added to accommodate increased load.

The Thor data model is a unique synthesis of ideas from programming languages and databases. Unlike many systems [Atkinson, 1983, Richardson, 1987, Weinreb, 1988, Nettles, 1992], it is language-independent rather than being embedded in a particular programming language. It provides access to object through both navigation and queries (as opposed to conventional database systems (e.g., [Stonebraker, 1990]) that support only queries). Furthermore, it supports full indexing for queries over sets of abstract objects, as opposed to the more limited techniques in [Bertino, 1989, Maier, 1986, Stonebraker, 1990, Zdonik, 1988]. Concurrency control and recovery are provided for individual objects (unlike many other systems

such as [Atkinson, 1983, Haskin, 1987]); furthermore programmers can control the granularity of concurrency control and recovery (the techniques in [Weihl, 1985] are being extended here). Unlike all other systems, it provides highly-available access to persistent objects. Finally, its implementation is fully distributed; not only are clients separated from servers, but the servers themselves are distributed.

Thor is an ongoing project, and this paper is a snapshot: we describe our first design and a partial implementation of that design. That design emphasizes issues related to the implementation of an OODBMS as a distributed system. We are currently working on a second design and a different implementation.

In the next section, we sketch the basic system architecture. Subsequent sections discuss the elements of this architecture in more detail.

2 SYSTEM ARCHITECTURE

Thor is intended to run in an environment of computing nodes connected by a network. Some of these nodes are Thor servers, which store the objects in the Thor universe. Others are client nodes where users of Thor run their programs. Although it is possible that a single node might act as both a server and a client, the common case is for client and server nodes to be distinct; we assume this separation in the remainder of this discussion.

The Thor system runs on both clients and servers. It runs front ends (FEs) at the client nodes, and back ends (BEs) and object repositories (ORs) at the servers. Users always interact with Thor via an FE, which typically resides at the user's workstation. The FE makes use of BEs and ORs to carry out client requests. The FEs and BEs understand types and perform operations. The ORs are concerned only with managing the storage for the persistent objects.

Every persistent object resides at one of the ORs. Usually this is the OR selected when it first became persistent. An object can move from one OR to another under user control, although we expect this to be rare. Each OR runs at one or more servers, and its objects are replicated at those servers (see Section 2.4).

There are two major obstacles to achieving good performance in such a distributed system: (1) the delay inherent in having clients and servers at distinct nodes, and (2) high load at servers, which slows down their response to requests. Below we discuss our approach to overcoming these obstacles.

When a user asks the FE to perform an operation on an object, the system will not run well if doing the operation always involves a remote access, e.g., to the OR that stores the object. The delay is not so bad if the operation involves a significant amount of computation. More typically, however, operations will be short relative to the network delay, so we would like to avoid the delay.

The situation is especially difficult in Thor because we assume that there are many servers and the servers are geographically distributed. Although we expect most objects used in a client application to reside at a single OR that is physically close to the client node (e.g., on the same local area net), an application might use objects from several ORs, some of which are far away from the client.

Delay can be masked in one of two ways: by caching objects at the FE, so that operations can usually be executed at the FE without the need for remote access, or by combining a number of calls into a larger "combined operation" that can be performed at an OR, so that the cost of the remote access can be amortized across several calls.

The other performance issue is reducing the load at the servers. Work in distributed file systems has shown that servers are the critical resource in a distributed system like ours, and that the way to improve system performance is to offload work from servers to clients [Howard, 1988, Nelson, 1988]. The analog in our system is again caching at the FEs and running the operations there.

Therefore we do most client requests at the FEs. FEs cache copies of objects from the ORs. When a client executes an operation, the FE runs it, fetching objects from the ORs if necessary. We hope that fetching will usually not be necessary because the objects that are needed are already cached at the FE. We discuss techniques that improve the likelihood of hitting in the cache in Section 2.1.

In our first design, all operations run at FEs (with the exception of queries as discussed further in Section 2.5). Ultimately, however, we may run some operations at servers, in the BEs; this will allow us to use the "combined operations" method of improving performance. (For example, an operation to look up a multipart pathname could run at the BE.) We have yet to do a detailed design of BEs, so we do not discuss them further in this paper.

2.1 CLIENTS AND FRONT ENDS

In this section and subsequent sections, we present our first design for Thor and describe a partial implementation of that design (called TH and described in detail in section 3). To distinguish between capabilities that have been implemented and those that have only been designed, we use the present tense in describing the former and the future tense in describing the latter.

2.1.1 Starting Up

As mentioned, clients always interact with Thor via an FE. When a client program begins a session with Thor, the first step is to create or connect to an FE for the client. Each FE acts on behalf of a single client principal (although it could be used by several client processes). An FE will be authenticated, both in its own right, and on behalf of the client principal, to the ORs with which it interacts; we plan to use the Kerberos authentication service [Steiner, 1988] for this. Limiting an FE to a single client principal helps in satisfying our security requirement.

Usually the FE resides at the client workstation but it could run elsewhere, e.g., if the client machine is very small, or its system is inadequately secure. Typically, the client program runs in a separate address space from the FE, so that errors in the client program cannot cause the FE to malfunction. However, if client programs are written in a type-safe language, such as ML [Milner, 1990] or CLU [Liskov, 1984], we can run them in the same address space as the FE. Such an organization will provide better performance because it avoids the cost of inter-process calls. To reduce delay when the client and FE run in different processes, we plan to investigate

client "combined operations" as a way of amortizing the cost of the FE call over a larger unit of work. Combined operations might be extracted from the client code by means of a preprocessor.

As part of starting a session, a client will be able to indicate an initial "setup." This setup will initialize the cache at the FE to contain useful objects (e.g., a part of the database for the client application and implementations of the types used by the application) so that we can avoid a lengthy startup transient. We are investigating means for users to indicate what such a setup should be. The client would not be delayed until the entire setup arrives at the FE; rather, the FE will use the setup information to prefetch information in the background.

2.1.2 Computation at the Front End

The client program interacts with Thor by executing Thor commands, e.g., to start and terminate transactions, and to run operations. In our prototype, each client transaction is carried out at a single FE; we plan to investigate multi-FE transactions later, and we will also look at the question of having Thor be a player in a larger environment in which transactions span many databases.

Client programs typically refer to objects by means of *handles*. These are names created by the FE and given to the client program whenever an operation called by a client returns a reference to an object. Handles are meaningful only with respect to an FE, and only for the duration of the current session.

When a client calls an operation, the FE begins by doing type-checking: the arguments (handles and values) must be of the types expected by that operation. This type checking takes account of the type hierarchy: if a formal is of type T, the corresponding actual argument must be of some type S, where S is a subtype of T. Then the FE runs the operation; that operation may call other operations, but these calls do not need to be type-checked because the checking was done at compile time.

2.1.3 Interacting with the Object Repositories

In running the operation, the FE may discover (in a way that we describe shortly) that some required item (e.g., an object, or the code of an operation) is not in its cache. In this case it must fetch that item from an OR. A fetch returns a *block* containing the requested object and also a number of other objects that are related to it. In other words, we do not fetch single objects, but rather we fetch them in groups.

Fetching in groups is important because it can enable us to "prime" the cache by filling it with objects that are likely to be used shortly; it also gives us a way of dealing with small objects effectively. One of the major challenges of the implementation is getting the blocks right so that the objects brought over really are likely to be used. We are planning to investigate a number of *prefetching* strategies. Some of these are fairly static, e.g., when an object is stored at an OR, it is stored close to related objects on disk; a block could thus contain objects that are close to one another on disk. An example of a more dynamic strategy is to have the FE indicate in the fetch command other items of interest, e.g., if the required object x refers to

n others, the FE could indicate that the ith and the kth are also of interest. An interesting research question is how an FE might know what other objects are of interest: does this information come from a client combined operation, or is there some other way that a client, or a Thor program (coded in our implementation language), indicates this?

A block need not be sent to the FE as a single unit, but rather it can be streamed across (as in Mercury [Liskov, 1988]). As the objects arrive at an FE, they are copied into the FE virtual memory space, and any object references they contain, which are expressed in OR terms as *orefs* (see Section 2.2), are changed to FE virtual memory addresses, i.e., we are "swizzling" pointers. Our scheme here has similarities to Mneme [Moss, 1990] and also to other object-oriented systems such as [Kaehler, 1990]. If we encounter a reference to an object that does not reside at the FE, we create a *surrogate* for it.[1] The surrogate contains within it the object's OR and oref so that if we need to fetch it later, we know where to find it. A surrogate is only large enough to hold the information to find the real object, not large enough to hold the object. When an object for which there is a surrogate arrives at the FE, we modify the surrogate to refer to the object; this link will be snapped at the next FE garbage collection.

Using a surrogate is what causes the FE to "discover" that an object is not present in the cache, causing an "object fault" to occur. The FE will be able to avoid such faults by prefetching. For example, if it creates a handle for an object, and that object is a surrogate, it will be able to prefetch the object, its type information, and its code, if necessary, since it is reasonable to assume that these items will be needed shortly. This prefetching will be able to take place in the background, without incurring a delay that is visible to the client.

As operations run, they may create new objects. Some of these objects never become persistent; they are used during the session only as temporaries. Such objects reside entirely at the FE. Other new objects become persistent (by having references to them stored into persistent objects). Newly persistent objects are sent to ORs when the transaction that made them persistent commits. Unless specifically instructed otherwise at commit time, Thor stores each newly persistent object at the OR of the object that was modified to make it persistent, and stores the new object "close" to the modified object or components of the modified object so that subsequent fetches will work well.

Non-persistent objects have neither orefs nor oids (see Section 2.2); if the objects become permanent, orefs and oids are assigned at the ORs during the commit where the objects become permanent. The result of a successful commit includes information allowing a front end to fix up its copies of those objects to include the new data.

The roots of the FE are the handles that have been given to the client. The client can optionally inform the FE of its current set of handles, either by listing

[1] A surrogate in Thor is similar to a proxy object in Bellerophon [Dickman, 1992] although some details of implementation are different because Bellerophon and Thor make different assumptions about the structure of the underlying distributed system. A Thor surrogate is also conceptually similar to a surrogate as used in [Fang, 1992] although again the details are different.

the handles it still has or by listing the handles that it has determined it will not use in the future. When so informed, the FE runs a garbage collector that discards all objects not accessible from the client's current handles. The FE will eventually have a true garbage collector that runs whenever there is no more free space. Since prefetching may bring over large numbers of reachable but unused objects, reclaiming inaccessible objects may not free enough space at the FE. We may also need to *shrink* (that is, convert to surrogates) some accessible objects that have not been used recently. We are investigating policies governing what objects to shrink or discard as part of our work on prefetching strategies [Day, 1992].

2.2 OBJECT REPOSITORIES

The ORs provide storage for persistent objects. They send copies of objects to FEs in response to fetch commands, and they commit transactions in response to FE commands as described in Section 2.3. Periodically, an OR will do garbage collection and discard any inaccessible objects as discussed below.

2.2.1 Fetching and Segments

An OR's persistent objects are stored on disk, which is organized into *segments*. Segments are variable length; each will be allocated contiguously on disk. Each segment is intended to contain a group of related objects, i.e., a *cluster* [Gruber, 1992]. When asked to fetch an object, an OR responds by sending a block, as previously described. Currently a block (the unit sent) is identical to a segment (the unit stored), but we plan to experiment with more sophisticated mappings between segments and blocks.[2]

Our goals for the fetch command are (1) to limit the number of disk accesses required to carry it out, and (2) to respond with a block of related objects so that future fetches can be avoided. A fetch command identifies the required object by giving its oref. An oref is a name local to that particular OR. We divide an oref into two parts: a segment id, or *sid* for short, and an object number. The sid is used to locate the disk storage for the object's segment by looking it up in the sid-table; we hope to keep this table in primary memory. Then the entire segment is read from disk (unless it is already in primary memory). The segment includes a table mapping the object number to an offset within the segment. This second level of mapping allows an object to move within the segment and change size without affecting its oref.

We can ensure that at most one disk read per fetchis needed by simply stopping the construction of a block at any time that another disk read would be needed. As long as we have the object that was actually requested, we need not send any other objects. Note that the OR will be able to keep information about relationships between segments; once a segment has been fetched, it will be able to prefetch a related segment from disk so that a subsequent fetch may be able to avoid any disk accesses.

Disk performance at the OR will improve if objects are clustered well. Quite a bit of

[2]Building a block from one or more segments is similar to *assembly* as described in [Maier, 1992].

work has been done on clustering strategies [Benzaken, 1991, Chang, 1989, Cheng, 1991, Shannon, 1991, Tsangaris, 1991, Gruber, 1992]. Our system is somewhat different from others, however, because objects will be written to segments in the background as explained in Section 2.4. Therefore, we will be able to reorganize segments on the fly if this turns out to be a good thing to do. We plan to experiment with techniques for evaluating the placement of objects in segment and reorganizing segments (for example, as in [Palmer, 1990]).

2.2.2 Mobility and Identity

As mentioned earlier, every object has a unique identity. As long as an object stays at the same OR where it first became persistent, the ⟨OR-id, oref⟩ is a sufficient unique name (where the OR-id is a unique name for the OR). However, if the object moves (which we expect to be rare), it may be referred to by more than one such name (e.g., its name relative to the old OR and its name relative to the new one). To ensure that the relative names will resolve to the same object, we assign each object an OR-independent unique id called an *oid* when it becomes persistent.

When an object moves from one OR to another, it will leave behind an *OR-surrogate* at the old OR. This surrogate contains the ⟨OR-id, oref⟩ for the object at its new OR. A chaining technique like this is described in [Fowler, 1985]. We believe such a technique will be suitable for our system, both because we expect chains to be short, and because ORs will be highly available, so the probability of an FE being unable to access an object because an OR on the path to the object is inaccessible is very small. An alternative to the chaining technique that is also feasible is to use a highly-available location service [Hwang, 1987].

In addition to surrogates caused by mobile objects, references from an object at one OR to an object at another OR make use of OR-surrogates. Whenever an object containing a reference to such a surrogate moves to an FE, we will endeavor to prefetch the surrogate itself. The goal is to ensure that by the time the FE needs to follow the reference, it knows which OR to ask for the object.

2.2.3 Roots

Recall that persistence is based on reachability from the root of Thor. The root names a top-level directory that (conceptually) contains an entry for each OR. The top-level directory is not stored in Thor; instead we find ORs via external systems such as network name servers. Each OR contains a second-level directory.

The OR garbage collector will make use of three kinds of roots: the OR directory, a table for each FE that is currently using the OR, and a table for other ORs. The FE and OR tables contain lists of references used at the respective FEs and ORs. The garbage collector will discard any objects not accessible from these roots. FE references will be discarded when FEs terminate (at the end of client sessions), and also as a result of FE communications (e.g., after an FE garbage collects, it will inform ORs of what references it contains). To determine what to do with references from other ORs, we will use a distributed garbage collection technique. We are currently developing a distributed garbage collection algorithm based on [Hughes, 1985, Ladin, 1989, Shapiro, 1990]. We have not decided what local method to use, but we are investigating a generational scheme with several spaces at each OR.

The FE table that acts as a root for garbage collection must contain the orefs of all local objects that might be in use at the FE. In this way we will be able to avoid nasty surprises, e.g., in which an object relied on by the client is discarded by the FE garbage collector because it is thought to be persistent, and is also discarded by the OR garbage collector because it is no longer persistent (due to the activities of some other client). We will fill the FE table lazily, as transactions commit that could cause objects to become apparently unreachable [Maheshwari, 1992].

2.3 TRANSACTIONS

So far we have not discussed how we achieve the atomicity properties—serializability and totality—of transactions. Totality is easy to achieve: all modifications are made at FEs and are installed in phase 2 of two-phase commit [Eswaran, 1976] only if phase 1 is successful. Serializability is more difficult. The problem is that operations are performed at FEs, yet we need to coordinate concurrent transactions (from different FEs) at the ORs. One way to do this would be to have FEs set locks at the ORs, but this would obviate the advantages of FE caching since it would require communication with ORs even when there is a cache hit. Therefore we use an optimistic scheme [Kung, 1981] that works as follows.

Every persistent object has a version number v# that is advanced each time a transaction that modifies the object commits. Each object contains its current v#, which is copied to the FE along with the object. While a transaction runs at the FE, the FE keeps track of what objects it reads and modifies. When the client requests a commit, the FE sends the v#s of all the objects read by the committing transaction, and the v#s and new versions of all the objects modified by the transaction, to an OR.

If all objects used by the transaction reside at that OR, committing can be done locally there; otherwise we need to do two-phase commit. (To make it more likely that transactions concern just one OR, Thor stores objects at ORs where related objects reside.) This OR acts as the coordinator of two-phase commit; the participants are the ORs that manage the objects that were read and modified by the committing transaction. In phase one, each participant OR checks the v#s used by the committing transaction against the actual v#s of its objects; if any do not match, the transaction must abort. If all participants agree to commit, the coordinator selects a new v# larger than any of the current ones, and this is written into all the modified objects along with their new versions in phase 2 (this information is actually written to a log as explained in Section 2.4). Thus what we have is an optimistic scheme using backward validation. The FE is informed about the new v# in the reply from the coordinator; it can then update the v#s it has stored for the modified objects.

We hope to run two-phase commit without requiring any disk I/O. This will be possible because our replication algorithm allows us to keep log records in volatile memory (see Section 2.4). However, in phase 1 we do need access to the version numbers of the objects used by the transaction. If ORs have very large volatile caches, the necessary information is highly likely to be in primary memory; otherwise, v#s can be stored in the FE tables.

2.3.1 Reducing Aborts

Although this concurrency control scheme is correct, it has the problem (as do all optimistic schemes) of possibly leading to aborts that need not have happened in a pessimistic scheme such as locking. Aborts in our system can come from two places: stale data in caches, and actual conflicts between concurrent transactions. To solve the first problem, we use *invalidation*: the OR notifies FEs when objects in their caches become stale as a result of transaction commits. ORs use the information in their FE tables to determine what invalidations are required. The front end receiving an invalidation message shrinks the stale objects back to surrogates, causing them to be refetched if needed. If the objects have been read by an active transaction, the transaction is aborted before shrinking the objects.

We are looking at two ways to reduce the problems of concurrent transactions. Our plan in both approaches is to move from an optimistic scheme to a locking scheme when there is contention, because locking is known to work better in such a case [Agrawal, 1987]. The first possibility is to switch from optimistic concurrency control to locking for objects that are hot spots. The OR would determine when this happens (e.g., because there had been contention at the object) and how long the object continues to be "hot". For hot objects, FEs would request locks before doing operations. The second possibility is to do lazy locking, in which the FEs do not wait to obtain locks, but instead notify the ORs about them in the background. The information thus obtained could be used to abort transactions early when there are conflicts (so that less work would be lost), to change which transactions abort (e.g., we could abort a writer if its commit would invalidate a concurrent reader), and to order commits intelligently (e.g., delaying commits of writers if they would invalidate concurrent readers).

Note that we use objects as a basis of concurrency control rather than pages or segments as has been done in most other systems [Deux, 1990, Haskin, 1987] because using a larger granularity can lead to false deadlocks and spurious conflicts. In addition, we plan to provide implementors of Thor objects with the ability to control the granularity of locking, as is done in Argus [Weihl, 1985]. User control can lead to better performance by decreasing conflicts and improving concurrency. The scheme for indexes discussed in [Liskov, 1992] is an example.

2.4 REPLICATION

We are planning to use a primary copy scheme for replication [Oki, 1988]. This scheme is an adaptation of one that we have been using in our work on the Harp file system [Liskov, 1991]. Harp is a replicated Unix file system that is intended to be used via a network file server such as NFS. Our motivation for building the file system was partly that such a service would be useful and partly to make sure that our replication method would be satisfactory. Our performance measurements to date have been encouraging; our replicated system performs slightly better than an unreplicated system on standard benchmarks.

Our primary copy scheme will work as follows: As mentioned, each OR will reside at a number of servers; the OR's objects will have multiple copies at these servers. For each object, one of the servers that has a copy of that object will act as the

primary; others with copies will act as *backups*. Actually, objects will be distributed in units of segments; a server will act as a primary (or backup) for all the objects in a segment. The primary will handle all FE interactions (fetches and commits) for that object. It will maintain a log in which it records information about commits, e.g., the new versions and v#s of modified objects, and will force the log to the backups at appropriate places, e.g., at transaction prepare or commit. If the primary fails, the backups and other servers of the OR will carry out a view change [El Abbadi, 1985, El Abbadi, 1986] and one of the backups will become the new primary.

All servers will have uninterruptible power supplies (UPS's) in addition to disks; the UPS's protect against power failures, which are the most likely cause of a simultaneous server crash. The UPS's will allow us to view a record (such as a commit record) as safely recorded in the log as soon as it resides in volatile memory at the primary and backups; the log will be written to disk in the background. Thus forcing a record to the log will require one message roundtrip; since the primary and backups are typically located close to one another, this time is usually shorter than a disk write. Records will not need to be kept in the log indefinitely: only those records whose effects have not yet been recorded on disk at the primary and backups will need to be saved. Thus, the log typically will be short.

When there is a failure, the view change algorithm will select a new primary and backups for the object and will define the initial state of the new view. The initial state is guaranteed to contain the log records of all prepared and committed transaction. This guarantee follows from the fact that log records are forced at critical points (prepares and commits) and because views will always overlap in at least one member. Therefore, the needed log records will be known to at least one member of the new view. Other members of the new view will be brought up to date by sending them any of these records they do not already have. Since logs will be short (as discussed above), bringing the other members of the view up to date will not take long, so failover is fast.

When there is a recovery, the recovering node is likely to be very out of date (especially if it has been down for a long time). In this case we will bring it up to date before doing the view change, in parallel with running the current view. Only when its state is almost current will we do the view change; at this point little information needs to be exchanged and again we will have fast failover.

One of the interesting implications of our replication method is that it will speed up two-phase commit, since no writing to disk is required. In addition, since information in the log is moved to disk in the background, the OR will be able to take the time to organize segments as they are changed. For example, new objects can be added to the appropriate segments, and if objects in a segment grow, we will be able to reorganize the space to accommodate this. Therefore, we should be able to put the right objects together in segments and thus obtain the benefits of reading entire segments.

2.5 QUERIES AND INDEXES

The ability to identify objects by means of queries is an important property of Thor. Queries introduce two new problems: (1) how to maintain indexes when the set being indexed contains objects of abstract, user-defined types, and (2) where to

run the query.

Efficient index maintenance requires some way of recognizing when a particular change to an object indicates that a particular index must change. In a database system, changes that affect indexes are easy to recognize, since these are just inserts into indexed sets, and updates of tuples in indexed sets. When sets contain abstract objects, however, the system no longer knows what constitutes an update, and furthermore objects in an indexed set may contain references to other objects whose modifications can affect an index for the set. We are currently investigating techniques for solving these problems [Hwang, 1992]. Note that the solution must take account of distribution: Changes to objects happen at FEs but indexes are stored at ORs, and furthermore the right kind of concurrency control must be in place.

We are also investigating several ways of running queries. If the set being queried is large and the query results in a small number of selected objects, it is desirable to run the query at the OR. However, if the queried set is not large, or if the query results in lots of matches, or if many queries are run on the same set, it may be better to move the set (or possibly just its index) to the FE and run the query there.

Ultimately we plan to use a flexible approach in which the FE decides where to run the query based on information available to it at that time. When a set object is used at an FE, meta-data about the set is sent to the FE, but the elements of the set are not. The meta-data includes information about indexes so that the FE can make decisions about how to carry out the query most effectively. If the FE decides to run the query at the OR, it sends a "match" command to the OR indicating what index to use and what index value or values are of interest. The OR responds by creating a new set containing the object that matched and either sending a reference to it to the FE, or sending the new set to the FE. (It will be possible to stream the elements of the new set to the FE so that the FE code can start to use them before they all arrive.) Alternatively, the FE can request that an index be sent to it, or that the set elements be sent (again the elements can be streamed).

If the indexed set is distributed, i.e., its elements are stored at more than one OR, the index will be distributed so that it has a component at each OR where there are elements. The component will store the portion of the entire index that concerns the elements at its OR. This organization will allow the FE to do the "matches" at the different ORs in parallel; it will also allow for easy index maintenance as the set and its elements change.

3 STATUS AND PLANS

We have implemented a partial prototype of Thor, called TH. TH is implemented in Argus [Liskov, 1988b]. It is a distributed system in which clients run at different nodes than ORs, and there are several ORs. We have built a veneer for Emacs Lisp [Lewis, 1990] and Argus, and have written both a toy hypertext application and a toy mail system on top of Emacs and TH. Several members of our group use a shared calendar system (of local origin) called ical; we plan to implement a version

of this system to run on top of TH. We are currently designing veneers for C++ [Stroustrup, 1987] and CLOS [DeMichiel, 1987] and are working with groups who have other real applications to test TH.

Abstract types in TH are defined in a stylized dialect of Argus, and all objects are implemented in terms of a single underlying primitive object type called f_obj. An f_obj is a variable-length heterogeneous array: it is essentially an Argus array with tagged elements. A slot in an f_obj can hold an integer, boolean, character, string, or reference. There is also a field for "self" that allows an object to refer to itself before it has been allocated. References are constrained to occupy the same space at the FE (where they are 32-bit VM addresses) and at the OR (where they are 32-bit orefs). References that require more bits are done by an indirection through a surrogate, in a style similar to that of Mneme [Moss, 1990].

Argus stable storage was useful for getting the system up quickly, but our research version of Argus limited the size of databases that we could build in TH. We have moved most persistent store management into TH, delegating the details of disk storage to the Unix filesystem for now.

TH does not implement our full design. The following features of Thor are not yet in TH: replication, control of disk storage, real programming and specification languages for Thor types, security and access control, distributed garbage collection, queries, and indexes. In Thor, code is stored at the ORs and fetched to the FEs, but TH statically links code into FEs.

We plan to add distributed garbage collection, queries and indexes, and object mobility to TH. We also expect to compile our Thor language into TH abstract type definitions (that is, in one mode the compiler will produce the appropriate stylized Argus code as the object code). However, we do not plan to implement replication, disk management, or access control in TH. It seems unlikely that we will attempt to implement code objects in TH.

Implementing TH has allowed us to firm up and test some of our implementation decisions. We plan to use TH as a test bed for studying various research issues, e.g., prefetching mechanisms, indexing mechanisms, and distributed garbage collection techniques. TH will also be used as a basis for some applications so that we can get a better understanding of how well our design meets the needs of users.

We plan to begin implementing a full prototype of Thor soon. Performance and portability are important goals of this implementation. We are likely to implement it in either Modula-3 or C++. We will use Recoverable Virtual Memory [Mashburn, 1992] as our initial storage manager, then move to using our replication scheme.

We are also working now on the design of the Thor implementation and specification languages, and at various extensions to the Thor model. For example, we are looking at triggers and constraints, and considering support for very long transactions.

Acknowledgements

Thor is a joint research project carried out by the authors and a number of others. Members of this team include Atul Adya, Boaz Ben-Zvi, Dorothy Curtis, Sanjay Ghemawat, Robert Gruber, Deborah Hwang, Paul Johnson, Umesh Maheshwari,

and Andrew Myers. In addition, the design has benefited from conversations with Stan Zdonik and David Langworthy.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136 and in part by the National Science Foundation under Grant CCR-8822158.

References

Agrawal R., M. Carey, and M. Livny. (1987) Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4):609–654.

Atkinson M. P. et al. (1983) An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365.

Benzaken V. and C. Delobel. (1991) Enhancing Performance in a Persistent Object Store: Clustering Strategies in O2. *Book chapter in Implementing Persistent Object Bases — Principles and Practice, ed. A. Dearle, G. M. Shaw, S. B. Zdonik*, Morgan Kaufmann.

Bertino E. and W. Kim. (1989) Indexing Techniques for Queries on Nested Objects. IEEE Transactions on Knowledge and Data Engineering, 1(2):196-214.

Chang E. (1989) Effective Clustering and Buffering in an Object-Oriented DBMS. Technical Report UCB/CSD/89/515, University of California, Berkeley.

Cheng J. and A. Hurson. (1991) Effective Clustering of Complex Objects in Object-Oriented Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 22–31.

Day M. (1992) *Cache Management in a Distributed Object Database*. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (forthcoming).

DeMichiel L. and R. Gabriel. (1987) The Common Lisp Object System: An Overview. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 151–170.

Deux O. (1990). The Story of O2. IEEE Transactions on Knowledge and Data Engineering, 2(1):91-108.

Dickman P. (1992) The Bellerophon Project: A Scalable Object-Support Architecture Suitable for a Large OODBMS? *Book chapter in Distributed Object Management, ed. M.T. Özsu, U. Dayal, P. Valduriez*, Morgan Kaufmann [this volume].

El Abbadi A., D. Skeen, and F. Cristian. (1985) An Efficient Fault-Tolerant Protocol for Replicated Data Management. In *Proceedings of the Fourth Symposium on Principles of Database Systems*, pages 215–219.

El Abbadi A. and S. Toueg. (1986) Maintaining Availability in Partitioned Replicated Databases. In *Proceedings of the Fifth Symposium on Principles of Database Systems*, pages 240–251.

Eswaran K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger. (1976) The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624-633.

Fang D., J. Hammer, and D. McLeod. (1992) An Approach to Behavior Sharing in Federated Database Systems. *Book chapter in Distributed Object Management, ed. M.T. Özsu, U. Dayal, P. Valduriez*, Morgan Kaufmann [this volume].

Fowler R. J. (1985) Decentralized Object Finding Using Forwarding Addresses. Technical Report 85-12-1, Department of Computer Science, University of Washington.

Haskin R., Y. Malachi, W. Sawdon, and G. Chan. Recovery Management in Quicksilver. *ACM Transactions on Computer Systems* 6(1):82–108.

Fowler R. J. (1985) Decentralized Object Finding Using Forwarding Addresses. Technical Report 85-12-1, Department of Computer Science, University of Washington.

Gruber O., and L. Amsaleg. (1992) Object Clustering in Eos. *Book chapter in Distributed Object Management, ed. M.T. Özsu, U. Dayal, P. Valduriez*, Morgan Kaufmann [this volume].

Hughes J. (1985) A Distributed Garbage Collection Algorithm. In *Proceedings of the Conference on Functional Languages and Computer Architecture*, pages 256–271.

Howard J., M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, M. West. (1988) Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems* 6(1):51–81.

Hwang D. J. (1987) Constructing a Highly-Available Location Service for a Distributed Environment. Technical Report MIT/LCS/TR-410, M.I.T. Laboratory for Computer Science, Cambridge, MA

Hwang D. J. (1992) *Indexing for Fast Associative Access to Large Object Sets.* Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (forthcoming).

Kaehler T. and G. Krasner. (1990) LOOM—Large Object-Oriented Memory for Smalltalk-80 Systems. *Book chapter in Readings in Object-Oriented Database Systems, ed. Stanley B. Zdonik and David Maier*, Morgan Kaufmann.

Kung H. T. and J. T. Robinson. (1981) On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6(2):213–226.

Ladin R. (1989) *A Method for Constructing Highly Available Services and a Technique for Distributed Garbage Collection.* Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Lewis B. and D. LaLiberte. (1990) *GNU Emacs Lisp Reference Manual.* Free Software Foundation, Cambridge, MA.

Liskov B., R. Atkinson, T. Bloom, J. E. B. Moss, C. Schaffert, R. Scheifler, A. Snyder. (1984) *CLU Reference Manual.* Springer-Verlag, New York.

Liskov B., T. Bloom, D. Gifford, R. Scheifler, W. Weihl. (1988a) Communication in the Mercury System. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, pages 178–187.

Liskov B. (1988b) Distributed Programming in Argus. *Communications of the ACM* 31(3):300–312.

Liskov B., S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. (1991) Replication in the Harp File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 226–238.

Liskov B. Preliminary Design of the Thor Object-Oriented Database System. Programming Methodology Group Memo 74, MIT Laboratory for Computer Science.

Maheshwari U. (1992) Distributed Garbage Collection in a Client-Server Transaction System. Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (forthcoming).

Maier D. and J. Stein. (1986) Indexing in an Object-Oriented DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 171–182.

Maier D., G. Graefe, L. Shapiro, S. Daniels, T. Keller, and B. Vance. (1992) Issues in Distributed Object Assembly. *Book chapter in Distributed Object Management, ed. M.T. Özsu, U. Dayal, P. Valduriez*, Morgan Kaufmann [this volume].

Mashburn H. H. and M. Satyanarayanan. (1992) RVM: Recoverable Virtual Memory. School of Computer Science, Carnegie Mellon University.

Milner R., M. Tofte, and R. Harper. (1990) *The Definition of Standard ML.* MIT Press, Cambridge, MA

Moss J. E. B. (1990) Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems* 8(2):103-139.

Nelson M., B. Welch, and J. Ousterhout. (1988) Caching in the Sprite Network File System. *ACM Transactions on Computer Systems* 6(1):135–154.

Nettles S. M. and J. M. Wing. (1992) Persistence + Undoability = Transactions. In *Proceedings of the 25th Annual Hawaii International Conference on System Sciences.*

Oki B. M. and B. Liskov. (1988) Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 8–17.

Palmer M. and S. Zdonik. (1990) Predictive Caching. Report CS-90-29, Department of Computer Science, Brown University, Providence, RI

Richardson J. and M. Carey (1987) Programming Constructs for Database System Implementation in EXODUS. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, pages 208–219.

Shannon K. and R. Snodgrass. (1991) Semantic Clustering. *Book chapter in Implementing Persistent Object Bases – Principles and Practice, ed. S. B. Zdonik and D. Maier*, Morgan Kaufmann.

Shapiro M., D. Plainfosse, and O. Gruber. (1990) A Garbage Detection Protocol for a Realistic Distributed Object-Support System. Research Report 1320, INRIA, La Rocquencourt, France.

Steiner J. G. and C. Neuman, J. I. Schiller. (1988) Kerberos: An Authentication Service for Open Network Systems. Project Athena, MIT, Cambridge, MA.

Stonebraker M., L. Rowe, and M. Hirohama. (1990) The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering* 2(1):125–142.

Stroustrup B. (1987) *The C++ Programming Language.* Addison-Wesley, New York.

Tsangaris M. and J. Naughton. (1991) A Stochastic Approach for Clustering in Object Bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 12–21.

Weihl W. and B. Liskov. (1985) Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems* 7(2):244–269.

Weinreb D., D. Gerson, and C. Lamb. (1988) An Object Oriented System to Support an Integrated Programming Environment. *IEEE Transactions on Data Engineering* 11(2):33–43.

Zdonik S. (1988) Data Abstraction and Query Optimization. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 368–373.