The Language-Independent Interface of the Thor Persistent Object System

Barbara Liskov   Mark Day   Sanjay Ghemawat   Robert Gruber
Umesh Maheshwari   Andrew C. Myers   Liuba Shrira

February 24, 1993

Abstract

Thor is a new object-oriented database system being developed at MIT. It allows applications written in different programming languages, and possibly running on heterogeneous machines and operating systems, to share objects conveniently. Our goal is to provide safe sharing of objects with higher-level semantics than is typical for today's file systems and databases, while still providing good performance. This paper describes the interface of Thor and also discusses some of the implementation techniques we are using to achieve our performance goal.

1 Introduction

Most organizations run their applications on heterogeneous equipment. Not only do these computers differ in their hardware, but they frequently run different software, e.g., different operating systems, as well. In addition, applications may be implemented in different programming languages and more than one language might even be used within a single application. Nevertheless, applications may need to share information in spite of these differences.

Thor is a new object-oriented database system intended to support such sharing. It is intended to run on a heterogeneous collection of machines connected by a communications network. It provides a universe of objects that can be shared by programs written in different programming languages. Objects are persistent and highly-available (i.e., with very high probability, objects entrusted to the system are stored reliably and are accessible when needed). They are encapsulated and *active*: applications can access them only by invoking their methods and the calls execute inside Thor, which ensures that no violation of encapsulation is possible. Objects belong to types that determines their methods; users can define new types and types can be organized into a type hierarchy. Finally, Thor defines a persistent root for the universe

and automatically discards objects that are no longer reachable from the root, and users are able to control object sharing by limiting the methods that specific users can call.

This paper discusses the Thor interface, and also describes some the implementation techniques we are using to achieve good performance. Achieving good performance is especially challenging since Thor is intended to run in a distributed environment in which applications run at client machines and persistent objects are stored at servers. This paper focusses on the techniques we use at the client machines to improve performance. Unlike other systems, Thor uses object caching rather than caching pages or files. Since objects may be small, and a client call might access many objects, when an object is fetched to a client workstation, we also prefetch a number of related objects. We believe that object caching with prefetching will provide better performance than more conventional approaches because we can get more useful objects into our cache and therefore get better cache utilization.

In current practice, cross-language sharing of persistent objects is typically achieved by file systems and databases. File systems often treat objects as uninterpreted byte streams, and therefore inter-language sharing via file systems can be laborious and error-prone. Conventional databases provide a more complex object structure, but the semantics of inter-language sharing is often left undefined. In either case, the underlying representations of data objects are accessed directly, making it impossible to enforce consistency contraints, and making it difficult to change an object's internal representation. Also, it is awkward or impossible for objects to contain references to one another, and object deletion must be explicit, since any kind of garbage collection is impossible if references cannot be distingushed from data.

Most existing object oriented databases, such as ObjectStore [15], Statice [28], Iris [9], and Orion [14] are specific to a single language. Of the systems that support multiple languages, the approach in GemStone [2] is the closest to ours; programs written in different languages access persistant objects through a method-based interface. However, Gemstone violates object encapsulation to enhance query performance. The approach in O_2 [7] is very different; different languages can access O_2 objects and implement methods. However, even though such a language keeps its syntactic appearance, it is reimplemented to include the O_2 data model, in effect resulting in a semantically different language. Finally, unlike any object systems of which we are aware, Thor objects are highly available and reside at multiple, distributed servers.

The remainder of this paper describes Thor and its implementation. We begin in Section 2 by describing the Thor interface. Then in Section 3 we discuss the Thor implementation, focusing on what happens in the part of Thor that runs on the client workstations. We conclude with a discussion of the current status of Thor and our future plans.

2 The Thor Interface

This section describes how users access active objects in the Thor universe, what causes objects to become persistent, and how users can extend Thor by providing new types and implementations.

2.1 Sessions, Handles, and Values

Interaction with Thor takes place within a *session*. A session starts with the client asking for the *root* object of the system. Either the client or Thor may end the session. Thor will end a session if it is unable to communicate with the client workstation for an extended period; this can happen because the client crashes or because of a network problem (a partition).

When the session starts, the client receives a *handle* for the root object. A handle is a small opaque data structure that the client application uses to refer to a Thor object. It is meaningful only for one session: if a client stores a handle outside Thor, ends its session, and starts another session, the stored handle will be invalid, and Thor will refuse to treat it as an object reference.

Clients use handles to request Thor to invoke methods of object (i.e., they use them to do *navigation*. Method calls return additional handles, and also *values*, such as integers or characters. Values show up on the Thor interface in an *external representation* defined by their type; the client program can then map the value into some internal form that makes sense in the client programming language[11].

Clients can also invoke stand-alone procedures. Such procedures are used to create new objects, and also to do computations that operate on objects rather than belong to objects (e.g., a sort routine).

2.2 Transactions

Every interaction with Thor implicitly takes place within an atomic transaction. Transactions simplify the semantics of the system in the presence of concurrency and failures, but they

require that clients identify the points at which data structures are consistent and should become persistent. When the client reaches such a point, it attempts to *commit* its changes. The attempt may fail, in which case the transaction *aborts* and all of its changes are undone; the client can also abort voluntarily. Starting a session implicitly starts a transaction, and committing or aborting implicitly starts the next transaction. Thus, client applications need to identify only the end of a transaction, not the beginning.

2.3 Persistence by Reachability

Thor provides a persistent root of the universe. All objects reachable from that root are themselves persistent. Objects created during a session become persistent is already-persistent objects are modified to refer to them by a transaction that commits.

For safety, Thor objects are never explicitly deleted. Instead, garbage collection reclaims resources used by unreachable objects. In addition to the persistent root, handles also serve as roots of garbage collection. It is desirable (but not required) for a client application to notify Thor about handles it is no longer using. For example, if the client language is garbage-collected, its collector could notify Thor about handles still in use at the end of a collection.

2.4 Extending the Type Universe

Users can define new types and they can provide new implementations for existing types. We provide a compiler that processes such definitions and adds objects that describe them to Thor in a type library. The library provides mechanisms for browsing the objects in it. (The library is actually an application of Thor that we provide.)

Types can be organized into a hierarchy. A type can have zero or more supertypes. A subtype must provide the methods of each of its supertype with compatible signatures [4]; in addition, it should simulate the behavior of its supertypes as discussed in [23, 22]. We allow a subtype to rename the supertype's methods, so that if the supertype has a method named "bar", in the subtype we can choose to name the method "foo" instead. Renaming is useful to avoid the name clashes that might otherwise occur (e.g., two supertypes of a subtype have a method named "foo" but with different behavior).

New types and implementations must be defined by writing modules in a new programming language we are developing. The new language has many safety features that we believe are

crucial in a persistent object store (e.g., it is strongly-typed and garbage collected). By contrast, many client languages (e.g., C) are unsafe and allow program errors (e.g., dangling references) that could damage or destroy persistent objects if we allowed programs written in them to run inside of Thor. Our approach is similar to that of GemStone [2], with its GemStone DDL/DML; however, while that language is an adaptation of Smalltalk, ours is a new object-oriented language, influenced by CLU [19] and Modula-3 [3].

Programmers can ignore our programming language if they just use existing types and implementations but not if they want to define new types or implementations. Because we realize that programmers would prefer not to cope with another language, we are studying whether restricted dialects of client languages can also be used to define new types and implementations.

2.5 Queries

Users of Thor can access objects by running queries over sets. Sets are objects in Thor, and as such they can be used in the same ways that other objects are used: they can be shared, are represented to the client as handles, and so on. They provide methods for querying over their members (e.g., a select based on a predicate). The efficient implementation of these methods requires attention (see [12] for details), but queries do not have a special place in the interface.

2.6 Operations

This section summarizes the preceding discussion by describing the operations of the language-independent interface. Figure 2.6 shows the calls made by the client to Thor and also the information that flows back as a result of these calls. In general a call either provides results or it signals an exception indicating a problem (e.g., the *must-abort* exception for the call of commit.

2.6.1 Get-root and End-session

A client application begins by asking for the root of the system, providing some information to identify itself. Thor may use the information to determine the actual object given to the client as a root. The signal *no-root* is raised if the information does not allow Thor to determine an appropriate root for the client.

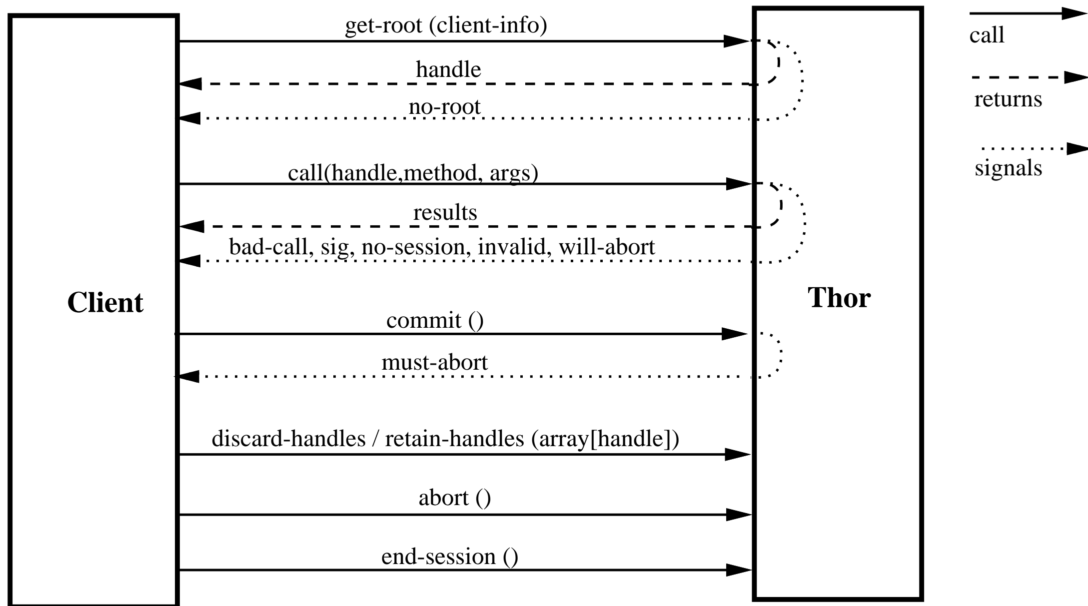

Figure 1: The language-independent interface

The end-session operation informs Thor that all resources associated with maintaining the current session may be discarded. The current transaction is aborted.

2.6.2 Call

To call a method, the client indicates the handle of the called object, the method name, and zero or more arguments. If the call is successful, zero or more results are returned. Both the arguments and results can handles and values. To call a stand-alone procedure, the handle identifies the procedure objet, and the method is "execute".

Thor checks that the call is type correct (the handle must be valid, and the object it identifies must have a method of that name, and must take the given number and types of arguments) and calls the method on the object if it is. The various signals indicate problems: *invalid* indicates a bad handle, *bad-call* occurs if the call is not type correct, *sig* reflects a signal raised by the called method, *no-session* indicates that Thor has closed the session, and *will-abort* indicates that the current transaction cannot commit because it has made use of an obsolete copy of an object.

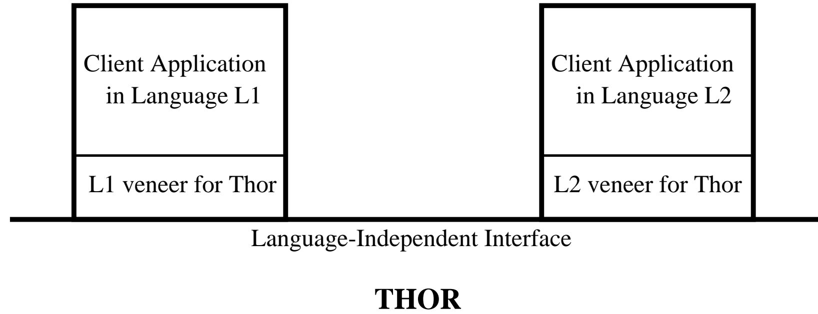

Figure 2: Applications, veneers, and the language-independent interface

2.6.3 Commit and Abort

The commit and abort operations end the current transaction. A successful commit installs all of the transaction's changes in Thor. An abort, either from calling abort or when a commit signals *must-abort*, undoes all of the changes of the current transaction. The creation of new objects is not undone (it is difficult to understand what a variable referring to such an "uncreated object" should mean), but any changes to that object are undone so that it returns to its state as first created. (This is the same semantics as in Argus[17].)

2.6.4 Retain-handles and Discard-handles

These operations inform Thor about handles no longer needed by the client. Retain-handles lists handles still in use by the client; discard-handles lists handles that will *not* be used subsequently. We provide both forms to match different storage management strategies in the client.

3 Implementation

The architecture of the implementation is shown in Figure 3. Part of the implementation (the *veneers*) sits on top of the interface; a veneer makes it easy to use Thor from a particular programming language. Below the interface are components that run at client workstations (the FEs) and components that provide reliable and highly-available storage for persistent objects (the ORs). First we briefly discuss the veneers, and then the interface implementation, focussing on what happens at the FEs.

3.1 Veneers

The Thor interface is intended to be used by compiler specialists to provide extensions to programming languages. The extensions, which we call *veneers*, provide access to Thor features in a way that is natural for the particular language. Application programmers make use of Thor by writing programs in their language of choice, extended by the veneer. A veneer is defined just once for a particular client language, probably in the form of a preprocessor for the compiler of the client language. These techniques used in veneers are similar to those used in heterogeneous distributed systems (e.g., [11, **?**, 18]).

To support method calls, the veneer must provide client-side handles and also a way of getting values. For the values, a veneer must define a language type for each value type and a way of mapping between the external representation of a value and the representation of that value in the language. For the handles, there might be a client-handle type H associated with a particular Thor type T, and a set of procedures, one for each method of T, that take a client-handle of type H as their first argument. The rest of the signature of the procedure is derived from that of the method in some straightforward way. For example, for values, the associated language type is used, and for handles, the associated handle type is used.

Client programmers use these veneer types in interacting with Thor. The veneer will provide a way for them to look up the veneer specifications for types of interest. For example, it might allow browsing of the Thor type library, but when a programmer looks at a type definition, he or she sees the veneer presentation of the type (i.e., the signatures of procedures that stand in for the methods). When calls to Thor appear in client programs the veneer compiler (or preprocessor) generates code to make the call to Thor, i.e., it implements the procedures of the client-side type. This is very much like a stub compiler in distributed systems [18, **?**].

The client program usually runs in a separate address space to ensure safety; for some safe languages we may allow the client and the FE to share a single address space. Since inter-process calls are usually fairly expensive, it would be good to avoid them by bundling several calls together. We are investigating ways of making such "combined operations"[**?**]; our approach is based partly on the *promises* of Mercury[18], and partly on the work of Stamos[**?**].

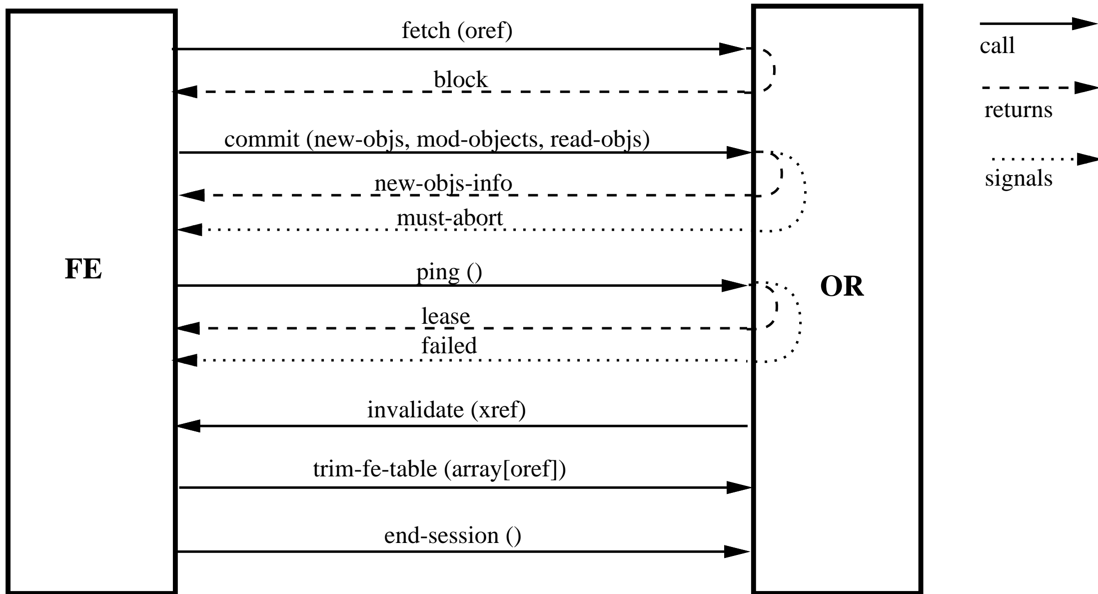

Figure 3: The FE-OR interface

3.2 Implementation of the Interface

Thor is a distributed system consisting of front ends (FEs) and object repositories (ORs). The ORs run on reliable server machines and provide persistent storage and concurrency control. The FEs implement the language-independent interface and run on client machines.

Each object resides at a single OR at any point in time, but can move among ORs; ORs track moving objects and participate in distributed garbage collection of persistent objects that are no longer accessible from either the persistent root or a handle at an FE. ORs are replicated for high availability and performance using the same technology as the Harp file system[20].

FEs are responsible for running client calls, communicating with ORs as neeeded. To make calls run fast, FEs make use of three techniques: caching, prefetching, and swizzling. They use the ORs via the interface shown in Figure 3.2: they fetch and prefetch objects into their cache by calling the fetch operation, and commit transactions by calling the commit operation. The remainder of this section focusses on the FE implementation. Details of the OR implementation can be found in [21].

3.2.1 OR-Sessions

When an FE first contacts an OR, say to fetch an object, it opens an OR-session with the OR. during the session, the FE and the OR maintain information about each other. At the OR there is an *FE-table*, which is a conservative record of the references held by the FE to objects at that OR.; the table is a root of garbage collection at the OR. When the client closes its session with the FE, the FE closes its OR-sessions.

Thor guarantees that all objects accessible to a client (via a path from a handle) will not be deleted by garbage collection. This guarantee is provided by *leases* [10]. An OR guarantees to maintain a OR-session with an FE for a limited time: until the lease expires. Normally, the FE renews the lease before it expires by sending ping messages in the background. In the event of a network partition or a crash of the FE, the lease will expire at both the FE and the OR. Ultimately the OR will close the session, discarding its FE-table, although it is very slow to do so because doing so means the client session must be closed because some references accessible from the client's handles may no longer be valid. Meanwhile the FE must stop processing client requests (since it cannot be sure that the OR still retains its OR-session information), although it can delay closing the client session in the hopes that it will reestablish connection with the OR and discover that the OR has not closed the OR-session.

More information can be found in [24]. It allows client sessions to survive failovers at ORs and avoids the need to maintain session information on stable storage at ORs.

3.2.2 Buffer Management

Unlike other systems, Thor manages its buffers in terms of objects. Rather than bringing in a page when an object is fetched, we bring in the requested object, together with some other objects that are related to it. Objects provide a clean basis for concurrency control and prefetching. Although DeWitt et al.[8] showed that interaction in terms of objects led to poor performance compared to fetching pages, we believe that it is misleading to compare a system that fetches single objects to a system that fetches single pages. The results of DeWitt et al. do show the necessity of prefetching and transferring groups of objects, rather than single objects. In Thor, we can transfer page-size groups of objects if we so choose; but we can also transfer groups that are larger or smaller than pages, and we can choose those groups based on

the current interconnections of objects and the identity of the client application requesting an object.

Our current plan is to follow pointers from the referenced object down some number of levels (perhaps two) and prefetch all objects that are at the same OR as the requested object and resident in the primary memory of that OR. (Furthermore, all these objects are highly likely to be resident in the OR's primary memory, as discussed in the next secton.) We believe that this strategy is more likely to prefetch relevant objects than simply bringing over an object's page. We plan to experiment with this and other schemes later, perhaps including schemes where the application can control prefetching. We are also exploring schemes for managing the FE buffer [5].

An FE cache contains both copies of persistent objects and newly-created objects that have not yet become persistent. Newly-created objects are not persistent when first created and might never become persistent, in which case they will never be sent to ORs. By keeping non-persistent objects entirely at FEs, we can avoid unnecessary expense; we effectively have a generational garbage collector where the youngest generation is transient and exists only at the front end.

3.2.3 Location-Dependent Names

We use location-dependent names to refer to Thor objects. Recall that each persistent object resides at a particular OR, although it may move to a different one. An object's name depends on its current OR. The name is an *xref*, which is a pair ⟨OR, oref⟩. The *oref* is a name local to that OR.

If an object at some OR R refers to another object at R, it does so by using just the oref of that object. Orefs can be relatively small, e.g., 32 bits. Therefore, we do not need to allocate very much space for these local references. Xrefs are of course much bigger, and cannot fit in the space allocated for an oref. Therefore, objects cannot refer directly to remote objects. We discuss how remote references are handled in Section 3.2.8.

An important aspect of our naming scheme is that fetching an object typically requires one message round trip and at most one disk read. Only one round trip is needed because the FE usually knows what OR to go to; this point is covered in the following sections. The OR can

usually fetch objects with at most one disk read because the oref actually contains two parts, a segment number and an object number relative to that segment. The segment number is looked up in a table that is small enough to be kept in primary memory. Therefore, the only disk read needed is to get the segment. In fact even that read is usually not needed because objects are clustered in segments and the OR maintains recently used segments in its cache.

Our scheme for object naming is discussed in more detail in [6].

3.2.4 Objects and Surrogates

When an object is copied to an FE, it is desirable to change its pointers to other objects to direct memory pointers at the FE, i.e., to virtual memory addresses. If we make this change then following pointers at the FE will be inexpensive; otherwise it would be expensive since we would need to look up orefs and interpret them. ((?ref to loom or ooze – one of these did this)) Changing orefs to addresses is called *swizzling* **??**.

In a scheme that uses swizzling, some way of handling pointers to objects that are not in the cache is needed. This can be accomplished using either *edge marking*, in which the form of the pointer indicates whether the object is present in the cache, or *node marking*, in which all pointers refer to local objects, but some of them just represent non-resident objects. We use the latter scheme; we call the non-residents *surrogates*. A surrogate contains its object's xref. It is like a forwarder in Mneme [26] or a leaf in LOOM [13].

Here is what happens when a group of objects arrives at an FE:

1. The objects are entered into the *swizzle table*. The swizzle table contains entries for every object at the FE, mapping between xref of the object and its current virtual memory address.

2. The requested object is swizzled: all its pointers to other objects are changed from orefs to virtual memory addresses, by looking in the swizzle table. If an object referred to isn't at the FE (and therefore not in the swizzle table), we create a surrogate for it, enter the surrogate in the swizzle table, and swizzle the pointer to contain the virtual memory address of the surrogate.

After these two steps, we have a fully-swizzled copy of fetched object in the FE's virtual memory,

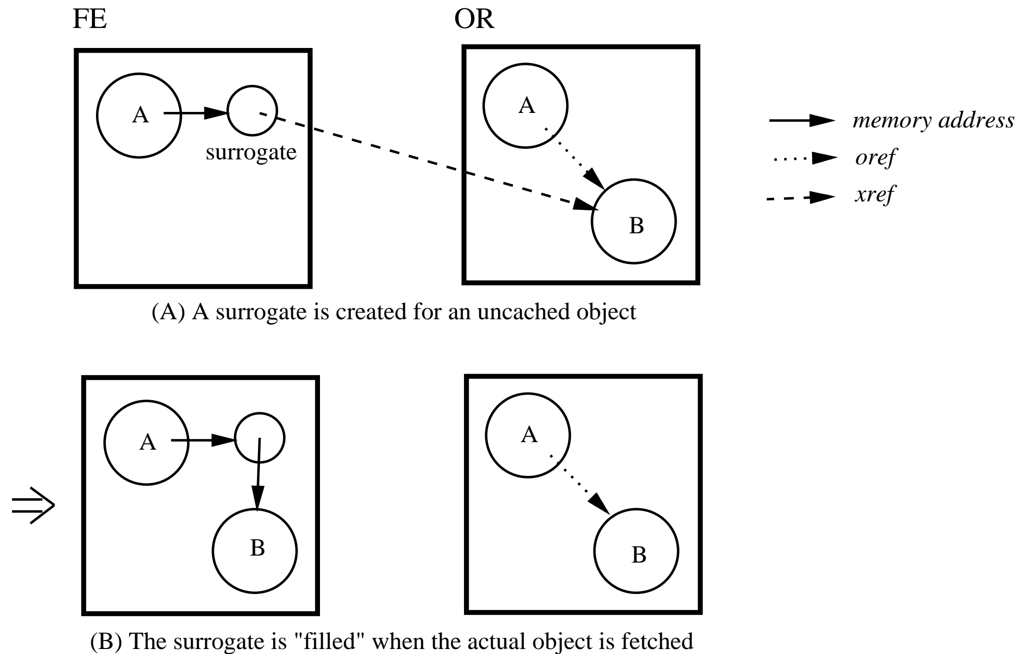

Figure 4: The FE-surrogate for B is created, then filled

some number of prefetched objects registered in the swizzle table, which can be individually swizzled on demand if some prefetched object is used later, and some surrogates that can be *filled* if needed. When a method of the surrogate is called, the FE fetches its object from its OR (using the xref in the surrogate to determine the OR), stores a pointer to the object in the surrogate, and passes control to the code of the called method. An extra level of indirection is introduced by each filled surrogate, but the link is snapped the next time the FE does garbage collection. The situation is illustrated in Figure 3.2.4.

Thor can *shrink* an unmodified persistent object at any time, replacing the object with a surrogate. One reason for such shrinking is to free space in FE buffers. Another, described in section 3.2.6, is to remove an outdated version of an object from the cache.

3.2.5 Method Dispatch

Invoking methods at the FE can be made fast because our language is statically typed, allowing a dispatch technique similar to that used in C++. The fixed set of methods understood by a type is mapped statically onto small integer indices. An object contains a pointer to a method dispatch vector, each entry of which points to code for a method, at a statically-determined

index.

A surrogate object has a method dispatch vector like any other object, but it contains special methods that fill the surrogate if they are called. Thus, every call is dispatched in the same way, and code of regular methods need not examine the object to determine whether it is a surrogate. Therefore, we incur no additional expense for surrogates for regular (non-surrogate) objects.

We also use this dispatch vector technique for the prefetched (unswizzled) objects. The dispatch vector for such an object points to special methods that first do the swizzling, then update the dispatch vector to refer to the regular methods. This *lazy swizzling* avoids the filled surrogate indirection. Also, eager swizzling would spend time swizzling many objects that are prefetched but never used [25]; lazy swizzling avoids this overhead.

We are working on schemes that allow dispatching to be replaced by direct calls in the presence of swizzling [27].

3.2.6 Transactions

The calls that make up a transaction run using objects at the FE, and fetching additional objects if necessary. If a call hits in the cache, we do not want to communicate with an OR to obtain a lock, since that would undo much of the advantage of prefetching. However, we cannot lock prefetched objects, since we don't know at that point whether they will even be used, nor what the mode of use (read or write) will be. Therefore we use an optimistic concurrency control scheme. Every object has a current version number. When a transaction tries to commit, if it used an obsolete version of an object, the current version number of that object will have changed, indicating that the transaction must abort.

The FE keeps track of which objects have been read or modified by the current transaction. When the client application asks for the current transaction to commit, the FE uses this data to assemble the information it needs to send to the ORs: the version numbers of all objects that were used, the new state of each modified objects, and the new states of any newly-persistent objects. The newly-persistent objects are found by tracing references from the modified objects. If a reference is found to an object not listed in the swizzle table that object does not exist at an OR, and therefore is about to become persistent.

Both the modified objects and newly-persistent objects must be unswizzled before being sent back to the ORs. The FE uses the swizzle table to do this. References to newly-persistent objects must be treated specially since the ORs create their names, not the FE; they are unswizzled into an index representing the object's sequence number in the list of newly persistent objects for this transaction, and they are tagged to distinguish them from orefs.

The FE sends the commit information to an OR (one that owns some of the objects used by the transaction). That OR acts as the coordinator of a two-phase commit process in which ORs that own objects used in the transaction attempt to validate the transaction (i.e., check whether the versions used are still the current versions). (A faster commit is used if only that OR is involved in the transaction.) More information about our commit technique can be found in [1].

If the commit fails, the FE is given the xrefs of all the objects for which validation failed. It discards these objects (turning them into surrogates) and undoes all modifications made by the aborted transaction. If a discarded object is needed by a subsequent transaction, the latest version will be fetched from the OR at that point.

Our scheme ensures that a transaction commits only if it accessed the current versions of objects. If some cached data is stale, no inconsistent information will be made persistent, but time can be lost if transactions must abort because they used stale data. To prevent this loss of time, the ORs involved in a commit notify FEs about objects that have been invalidated by the commit. In response to such a message, an FE discards the invalidated object, turning it into a surrogate. In addition, if the object has been used by the current transaction, the FE aborts that transaction (this is the source of the *will-abort* exception seen in section 2.6).

3.2.7 Garbage Collection

Periodically the FE does garbage collection to free up space in the local cache. The roots for this collection are the handles in the handle table. Both the current and old versions of the modified objects must be included in the roots so that the proper objects will remain accessible whether the current transaction commits or aborts.

As noted in Section 3.2.1, an OR records the references held by an FE in the FE-table. The local garbage collection at the FE may get rid of many of these references. Therefore, after

a collection, the FE informs ORs of the references it continues to hold for their objects (the message is denoted as `trim-fe-table` in Figure 3.2). This information is used by the OR to remove entries from the FE-table, which is one of the roots for its garbage collection.

3.2.8 Inter-OR references

Now we discuss how to handle remote references at the ORs. Since an xref cannot be stored in a slot that is just big enough to hold an oref, objects cannot contain direct references to remote objects. Instead, we use *OR-surrogates* for such references. An OR-surrogate is much like the previously-described surrogates (which are properly called FE-surrogates) in that it contains an xref and serves as a reference to an object usually stored elsewhere; however, an OR-surrogate does not have a dispatch vector. When an OR-surrogate arrives at an FE, it is immediately converted into an FE-surrogate as part of the process of building the swizzle table.

When an FE fetches an object A containing a reference to a remote object B, it gets the oref of the surrogate for B that is stored at the OR holding A. This situation is illustrated in Figure 3.2.8. If the FE needs to use B, it must first obtain the xref stored in the surrogate, and then use the xref to fetch object B. This sequence of events implies that fetching object B requires two round trips. However, in the common case where the surrogate for B is in the OR's memory when a request for A arrives, we can eliminate the first round trip by sending the xref for B along with A, effectively prefetching the surrogate for B to the FE. With prefetching, there will usually be only one round trip to find an object, unless the object has moved from B. Even when the object has moved away, there will be a problem only if the move is very recent, because ORs communicate about objects that move, and update the xrefs stored in their surrogates when they find out about moves.

4 Status

We have implemented a partial prototype of Thor, called TH. TH is implemented in Argus [17]. It is a distributed system in which clients run at different nodes from ORs, and there are several ORs. We have built a veneer for Emacs Lisp [16] and Argus, and have written a toy hypertext application on top of Emacs and TH.

We have implemented the swizzling and filling techniques described, and local garbage

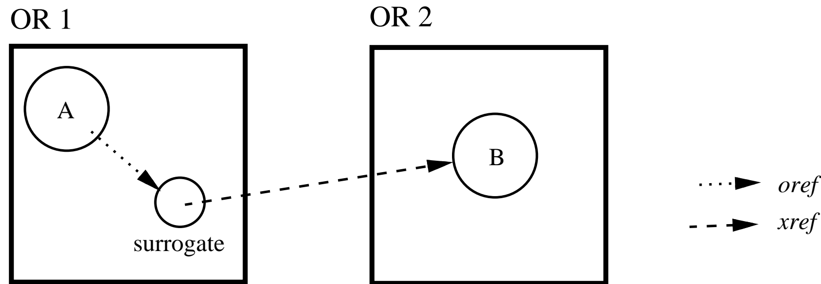

Figure 5: An OR-surrogate

collection built on top of the Argus garbage collector. Argus stable storage was useful for getting ORs working 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. Distributed garbage collection is currently being implemented.

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 are also using simulations as a way of investigating new techniques for method dispatching, reading and writing persistent storage, cache management, and concurrency control.

5 Discussion

Thor is a new object-oriented system intended to support heterogeneous sharing of objects in a distributed system in which clients run on workstations and persistent objects are stored at servers. Thor provides a universe of active objects; clients interact with Thor by requesting it to run methods on the shared objects. Such an interface guarantees that objects are used properly (i.e., by invoking their methods) and it also supports heterogeneity since objects can be freely shared between applications without knowing anything about what languages were used to implement the applications.

To obtain this safety and convenience at a reasonable cost, however, is a challenge. We

discussed some of the implementation techniques that are needed to get good performance. We focussed on the component of the system that implemented the Thor interface at the client workstations. The component (the FE) uses object caching and prefetching to improve performance of client code.

Thor is unique in managing the cache on the basis of objects rather than larger units such as pages and files. We compensate for the relatively small size of objects by bring over extra objects (prefetching) whenever we fetch an object. We conjecture that object caching will work better than caching based on larger units because we will be able to use the cache more effectively. We will be able to prefetch useful objects (more useful than just getting what is physically near the fetched object) and when we discard objects we can treat them as individuals. However, at present this is just a conjecture and we are performing experiments to test its validity.

We are starting to work on a full prototype implementation of Thor; performance and portability are important goals of this implementation. We are also firming up the details of our language design, and we are studing extensions to the Thor interface, such as triggers, constraints, and support for very long transactions.

References

[1] Atul Adya. A distributed commit protocol for optimistic concurrency control. Master's thesis, Massachusetts Institute of Technology, 1993. Forthcoming.

[2] Paul Butterworth, Allen Otis, and Jacob Stein. The gemstone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[3] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Language definition. In Greg Nelson, editor, *Systems Programming in Modula-3*, chapter 2. Prentice-Hall, 1991.

[4] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[5] Mark Day. *Managing a Cache of Swizzled Objects and Surrogates*. PhD thesis, Massachusetts Institute of Technology, 1993. Forthcoming.

[6] Mark Day, Barbara Liskov, Umesh Maheshwari, and Andrew C. Myers. Naming and locating objects in thor. Submitted for publication, 1993.

[7] O. Deux and et al. The o@-(2) system. *Communications of the ACM*, 34(10):34–48, October 1991.

[8] David J. DeWitt, David Maier, Philippe Futtersack, and Fernando Velez. A study of three alternative workstation-server architectures for object oriented database systems. In *Proceedings of the 16th VLDB Conference*, pages 107–121, 1990.

[9] D.H. Fishman and et al. An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[10] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 202–210, 1989.

[11] Maurice Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[12] Deborah J. Hwang and Barbara Liskov. A new indexing scheme for object sets. Submitted to VLDB, 1993.

[13] T. Kaehler and G. Krasner. *LOOM — Large Object-Oriented Memory for Smalltalk-80 Systems*, pages 298—307. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[14] Won Kim and et al. Architecture of the orion next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[15] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[16] B. Lewis and D. Laliberte. Gnu emacs lisp reference manual, 1990. Free Software Foundation, Cambridge, MA.

[17] B. Liskov. Distributed programming in argus. *Comm. of the ACM*, 31(3):300–312, March 1988.

[18] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the mercury system. In *Proc. of the 21st Annual Hawaii Conference on System Sciences*, pages 178–187. IEEE, January 1988.

[19] B. Liskov and et al. *CLU Reference Manual*. Springer-Verlag, 1984.

[20] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proc. of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

[21] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in thor. In Tamer Ozsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*, pages ??–?? Morgan Kaufmann, 1993.

[22] Barbara Liskov and Jeannette Wing. Family values: A semantic notion of subtyping. Technical Report MIT/LCS/TR-526, M.I.T. Laboratory for Computer Science, Cambridge, MA, January 1993.

[23] Barbara Liskov and Jeannette Wing. Using extension maps to define subtypes. Submitted for publication, 1993.

[24] Umesh Maheshwari. Distributed garbage collection in a client-server, transactional, persistent object system. Master's thesis, Massachusetts Institute of Technology, 1993.

[25] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. Technical Report 90–38, COINS, University of Massachusetts - Amherst, 1990.

[26] J.E.B. Moss. Design of the mneme persistent object store. *ACM Transactions on Office Information Systems*, 8(2):103–139, March 1990.

[27] Andrew C. Myers. Optimizing method dispatch in a heterogeneous object repository. Master's thesis, Massachusetts Institute of Technology, 1993. Forthcoming.

[28] D. Weinreb, D. Gerson, and C. Lamb. An object oriented system to support an integrated programming environment. *IEEE Transactions on Data Engineering*, 11(2):33–43, 1988.