A Safe, Efficient Object Database Interface Using Batched Futures

by

Phillip Lee Bogle

July 1994

© Massachusetts Institute of Technology 1994

This work 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.

Massachusetts Institute of Technology
Laboratory of Computer Science

545 Technology Square
Cambridge, Massachusetts 02139

A Safe, Efficient Object Database Interface Using Batched Futures

by

Phillip Lee Bogle

Abstract

For many systems such as operating systems and databases it is important to run client code in a separate protection domain so that it cannot interfere with the correct operation of the system. Clients communicate with the server by making cross domain calls, but these are expensive, often costing substantially more than running the call itself. This thesis describes a new mechanism called batched futures that transparently batches client calls so that domain crossings happen less often, thus substantially reducing the cost. This thesis also presents performance results showing the benefits of the mechanism on various benchmarks. Batched futures are described in a context in which they are particularly applicable and useful: the client interface to Thor, a new object-oriented database. A secondary contribution of this thesis is the description of this safe, language-independent client interface to Thor.

Thesis Supervisor: Barbara H. Liskov
Title: N.E.C. Professor of Software Science and Engineering

Keywords: futures, batching, protection domains, cross-domain calls, remote procedure calls, RPC, distributed systems, context switch overhead

Acknowledgments

I am indebted to many people for their input and assistance in this thesis.

Barbara Liskov, my thesis supervisor, who directed me in a promising research direction, made numerous helpful suggestions along the way, and helped me structure my ideas clearly in the form seen here. I owe her a debt of gratitude not only for supervising this thesis but also for her lessons in how to conduct and present research. Any errors exist in spite of her.

My office-mates, Atul Adya and Quinton Zondervan, who were ready at a moment's notice to discuss an idea, share their own research with me, or comment on one of the innumerable drafts of this thesis. They made office 526 an interesting and entertaining place to be.

To Mark Day, for introducing me to the Programming Methodology group before I came to MIT and for introducing me to Thor when I joined the group, and to him and all the other members of the Programming Methodology group for offering invaluable feedback on the thesis: Andrew Myers, Sanjay Ghemawat, Robert Gruber, Deborah Hwang, Liuba Shrira, Paul Johnson, and Dorothy Curtis. Thanks also to James O'Toole and Kavita Baala for their input.

To my parents and my parents-in-law in Seattle and Bangkok, and my extended family around the world, for their continued guidance and encouragement.

Finally, a special thanks to Manjari Wijenaike, my wife, for the fresh perspectives, support, and friendship she has always provided.

Contents

1. Introduction **13**

1.1 Related Work 15

1.2 Roadmap 17

2. Object-Oriented Database Model **19**

2.1 OODBs Goals 19

2.2 Compromises of Existing Object-oriented Databases 21

2.3 Achieving Safety and Its Costs 22

3. Thor Client Interface **23**

3.1 Objects and Operations 23

3.1.1 Iterators 24

3.1.2 Exceptions 25

3.2 Types 25

3.2.1 The Subtype Hierarchy 27

3.2.2 Typechecking 28

3.3 Transactions 28

3.4 Garbage Collection 29

3.5 Discussion: Transmitting the Representations of Arbitrary Types 29

3.6 High-level Architecture 30

3.7 Safety issues 32

4. Veneers **35**

4.1 Database Commands 35

4.2 Stub Types 37

4.2.1 Stub Type Interface 37

4.2.2 Stub Type Implementation 44

4.2.3 Storage Management 47

4.2.4 Veneer Support For Iterators 49

4.3 Implementing New Veneers 51

4.3.1 Stub Generators 51

4.3.2 Different Type Systems 53

4.3.3 Coordinating Client and Database Garbage Collection 55

4.4 Language Independence Issues 56

5. Batched Futures **59**

5.1 Introduction to Futures 59

5.2 Example 60

5.3 Implementation 62

5.3.1 Representing Futures 62

5.3.2 Mapping Futures to Objects 62

5.3.3 Limiting the Size of the Future Mapping 65

5.3.4 Stub Object Storage Management 66

5.3.5 Shared Memory Optimizations 67

5.4 Other Benefits of Batching Calls 68

5.5 Exceptions 69

6. Experimental Results **71**

6.1 Performance model 71

6.2 Measured Performance: Best Case 72

6.3 A Less Favorable Case 76

7. Extensions **79**

7.1 Futures for Basic Values 79

7.1.1 Example 82

7.2 Batched Control Structures 83

7.2.1 The Meaning of Batched Control Structures 84

7.2.2 Restrictions on Batched Control Structures 85

7.2.3 Evaluating Batched Control Structures 87

7.2.4 Additional Benefits of Batching Control Structures 89
7.2.5 Comments 90

8. Conclusions and Future Work **91**

Table of Figures

Figure 3-1: Example Type Interface 26
Figure 3-2: Subtyping 27
Figure 3-3: Thor Architecture 31
Figure 4-1: Database Commands 36
Figure 4-2: Type Interface in Theta 38
Figure 4-3: Stub Type Interface in C++ 38
Figure 4-4: Example Client Routine 39
Figure 4-5: Basic Exception Handling 43
Figure 4-6: Stub object Exception Handling 44
Figure 4-7: Object Representation 45
Figure 4-8: Cross-Domain Method Calls 46
Figure 4-9: Example Stub Function 46
Figure 4-10: Smart Pointer Implementation 49
Figure 4-11: C++ Stub Function Skeleton 52
Figure 4-12: Lisp Stub Function Skeleton 53
Figure 5-1: Batching interrelated calls using futures 61
Figure 5-2: Representing Futures 62
Figure 5-3: Returning Futures 63
Figure 5-4: Mapping Futures to Objects 64
Figure 5-5: Remapping Futures 66
Figure 6-1: Values of t_c and t_d for some systems 73
Figure 6-2: Best Case Performance— Simple IPC 75
Figure 6-3: Best Case Performance— Thor, Local 75
Figure 6-4: Best Case Performance— Thor, Remote 75
Figure 6-5: OO7 Traversal Performance 77
Figure 7-1: Client Control Structures 83
Figure 7-2: Batched Iterator and Conditional Control Structures 84
Figure 7-3: Batched Calls and Control Structures 85
Figure 7-4: The Assign-Once Restriction 86
Figure 7-5: Batched Depth First Traversal 87

Chapter 1.

Introduction

An important problem in the design of client-server systems is the tension that exists between safety, performance, and clean interface design. Many systems would ideally be structured as distinct subsystems, each running in its own protection domain and communicating with other subsystems via fine-grained calls. Protection domains are desirable because they prevent an ill-behaved client from corrupting data structures, reading private information, or otherwise interfering with the correct operation of servers, increasing the modularity, security, and debuggability of the system.

Unfortunately, there is a significant performance penalty associated with crossing protection domain boundaries (for example, the context switch overhead in Unix). Even the best cross-domain call implementations are many tens of times slower than direct calls [Bershad 90]. Crossing a protection boundary on every call is prohibitively expensive for lightweight operations, because much more time is spent crossing protection domains than getting work done. The overhead becomes another order of magnitude worse when the communicating subsystems run on different machines and communicate over a network. The problem is not likely to disappear with advances in software and hardware technology: context-dependent optimizations in hardware such as superscalar processors, pipelining, and caching make domain crossings all the more expensive, while increasingly complex software systems make them all the more important.

This thesis presents a general mechanism, called batched futures, for reducing the cost of cross-domain calls. The basic idea is that certain calls are not performed at the point the client requests them, but are instead deferred until the client actually needs the value of a result. By that time a number of deferred calls have accumulated and are sent all at once, in a "batch." In this way we can turn N domain crossings into one, and user code runs faster as a result. Our mechanism makes the batching essentially transparent to client

applications and allows later calls to make use of the results of earlier calls. In addition, it can be used when the client and server run on the same machine or different machines.

For batched futures to be useful, the client and server should be separated by a boundary that it is expensive (but possible) to cross, and that crossing should have a significant cost component independent of the amount of data being transferred between the domains. For example, in Unix, processes can communicate with other processes using IPC (interprocess communication) mechanisms such as sockets or shared memory, but must pay the penalty of a context switch on each communication. Other systems might use even heavier-weight protection boundaries, such as running the client program on a separate machine and communicating with it over a network.

Batched futures are particularly useful for interacting with object-oriented database systems (OODBS). In principle, an OODB combines the rich object model of a modern programming language with the language independence, persistence, and secure sharing of a traditional database. In practice, most OODBS fail to achieve one or more of these goals. In the interests of performance, they sacrifice safety— running clients in the same protection domain as the data and giving them unrestricted access to the representations of objects in the database— and/or language independence, supporting only a single client language.

Thor is a new database that aims to reconcile the goals of safety and performance. This thesis presents a client interface design for Thor that allows safe sharing of database objects by applications written in virtually any client language. Because this design places clients in a separate protection domain from the database, each interaction between the client and the database incurs the overhead of a cross-domain call. We then show how batched futures can be used to help reduce the overhead of cross-domain calls, increasing performance without sacrificing safety.

1.1 Related Work

A great deal of work has been done on reducing the costs of transferring control between protection domains, for example, Bershad's work on lightweight remote procedure calls

[Bershad 90]. Such work has made significant progress, but despite intensive effort has not succeeded in eliminating domain-crossing overhead. Our approach is complementary to these approaches in the sense that it amortizes the remaining overhead by reducing the number of crossings.

Several past systems have taken advantage of batching to reduce domain-crossing overhead, for example Mercury call streams [Liskov 88b] and the X window system [Getty 90]. However, past systems have failed to deal adequately with the question of what to return to the client in place of the actual return value. X opts for the rather draconian solution of disallowing return values for batched calls. Under Mercury, the client receives a promise for the eventual result of the call, which at a later point it can explicitly claim to obtain the actual value. However, a promise is an inferior substitute for the actual result, because it cannot be used an argument to another remote call. Our approach improves on these past approaches both in its transparency to users and in its ability to batch interrelated calls.

Futures were introduced in the parallel programming language Multilisp [Halstead 85]. However, they were used there to mask caller-callee parallelism on a multiprocessor system, rather than to batch calls in a serial system, as our scheme does. Because parallel futures introduce unrestricted concurrency, they can change the semantics of the client program, which is never the case with batched futures. Used unwisely, they can also increase the running time of a program, because of the overhead of the parallel fork and the additional tag checks required to check for futures, whereas batched futures never increase client running time. In a parallel system, a future might occur anywhere a basic value is expected, so the tag checks for futures "infect" the entire system. In our system, it is always the case that a future is evaluated by the time it is needed, eliminating the need for tag checks throughout the system.

Barrera has independently developed a scheme that seems closely related to ours in [Barrera 93]. However, the two page workshop paper describing his scheme fails to deal with a number of important issues including:

- *space efficiency*- the design seems to require that the system maintain an arbitrarily large mapping for an indefinite period of time.
- *time efficiency*- the design has an $O(n^2)$ copying overhead for a batch of n calls
- *type safety*- the paper describes the results of batched calls as promises, which are a distinct type from normal values, but says that they can be passed as arguments as if they were normal values, which implies they have the same type as normal values. This apparent contradiction is not resolved.
- *performance results*- the results given are too sketchy to be useful; the paper gives no indication of the degree of batching required to obtain the claimed results.

This thesis attempts to provide answers to each of these questions.

Software-based fault isolation [Wahbe 93] is a scheme for inserting software bounds checks into untrusted client code to ensure that it does not access data outside of its own region of memory, thereby allowing it to run safely in the same protection domain as the database. Unlike our scheme, software-based fault isolation adds overheads to the client code even when the code is not communicating with the database; however, these overheads are small and software-based fault isolation is likely to outperform our scheme for clients that interact frequently with a server. The major disadvantage of the scheme relative to ours is that it requires modifications to the compiler or a sophisticated binary patcher, which the authors admit is difficult if not impossible to write for arbitrary, unannotated binaries. (The authors do suggest that by adding additional information to the object file format, the binary patching problem becomes feasible. However, this just transfers the necessary modifications from the compiler on to the linker, which is not necessarily any less of a difficulty if all that is available is the linked binary.) Software-based fault isolation is therefore more difficult to implement and, in a practical sense, less portable than our scheme, which can easily be implemented for essential any programming language without compiler modifications. Also, software-based fault isolation is obviously not applicable if the two domains are actually running on separate machines, whereas our

scheme is. In fact, our scheme yields the greatest benefits in just that situation, in which the domain crossing overhead is largest.

1.2 Roadmap

The remainder of this thesis is organized as follows.

- **Chapter 2:** Motivates the need for batched futures by describing object-oriented databases and the safety compromises that past databases have made to get good performance.
- **Chapter 3**: Describes the client view of the Thor database. Clients see Thor as a universe of persistent, encapsulated, strongly-typed objects. Atomic transactions ensure database integrity in spite of system failures and concurrent access. Garbage collection automatically deallocates objects that are no longer in use.
- **Chapter 4:** Describes a client interface design, called a veneer, that provides the full power of Thor to an arbitrary client language without compromising safety. The essential features of the model are a separate protection domain for clients and a representation for object references that conceals the actual object representation.
- **Chapter 5:** Explains the main contribution of this thesis, batched futures, and their implementation in the context of Thor. Batched futures require simple generalizations of the client object representation and data structures already maintained by the database.
- **Chapter 6:** Presents a performance model and the results of performance experiments from three systems: (1) a lightweight IPC system, (2) Thor in its standard configuration, and (3) Thor with the client running remotely. We find that batched futures yield performance increases up to 10 times if high batching factors can be obtained. Even under worst-case conditions in which only low batching factors can be achieved, we still measure performance increases of over 1.7 times.
- **Chapter 7:** Adds several extensions to the basic batched futures model that increase the amount of batching that is possible. One extension allows futures to be used for

calls that return basic values as well as those that return Thor objects. Another allows simple control structures to be batched along with calls.

- **Chapter 8:** Gives our conclusions and discusses several areas for future research.

Chapter 2.

Object-Oriented Database Model

In this section, we discuss some of the goals of object-oriented databases, in particular those of the Thor system, and why the safety provided by protection domains plays an important role in satisfying those goals

2.1 OODBs Goals

Object-oriented databases (OODBS) are intended to provide a level of storage more closely matched to the needs and semantics of modern applications than that provided by traditional filesystems and relational databases. In principle, they incorporate a number of features from object-oriented programming languages, while retaining the best features of traditional databases.

One important advantage over traditional databases is the ability to store object operations, implemented in a powerful programming language, as well as the objects themselves. This allows objects to be *encapsulated*, that is, accessed only through a predefined set of operations (methods) rather than by directly modifying their representation. For example, an object-oriented database could support a balanced tree type, with operations that maintain the invariant that the tree object is balanced. Encapsulation allows the database implementer to ensure that the database correctly captures the objects' semantics, and that clients do not accidentally overwrite objects by directly modifying their representations. As a result, objects entrusted to the database are less likely to be corrupted by the client applications that share them.

Another advantage is that an OODB directly supports pointers between objects, allowing it to capture the interrelationships in a network of objects. Relational databases, in contrast, force applications to "flatten" the network into a linear collection of records, and simulate pointers using objects IDs and associative lookups. Such translations are harmful

to performance, inconvenient, and error-prone: the translation into the database might not be correct, or when the information is accessed it could easily be misinterpreted. In addition, because a relational database cannot tell the difference between an object ID and, say, a part count, it cannot perform useful tasks that depend upon following object references (for example, garbage collection and prefetching.)

Finally, an OODB supports a hierarchy of types and subtypes. The type hierarchy describes common behavior among a set of related types, facilitates code reuse, and simplifies reasoning about code. In contrast, most traditional databases only support a fixed set of basic field types arranged into records.

At the same time, the ideal OODB maintains the best features of traditional databases. In particular, the data in an object-oriented database should be *safely* shareable between client applications written in many different languages. We want to extend the full benefits of the database and object-oriented programming to any client language, even specialized languages with fixed type and data models such as Perl. Furthermore, we want that access to be safe, even if the client languages themselves are unsafe. Applications written in C++ and Lisp, for example, could both manipulate a shared set of objects, with the guarantee that neither one could violate encapsulation and corrupt objects.

An OODB also shields clients from many of the complexities of concurrency and system failure. It achieves these goals by allowing clients to group sets of related operations into transactions. Transactions are *atomic*: either all of the operations within that transaction take effect, or none of them do. They are also *serializable*: the effect of running a set of transactions is equivalent to running them in some serial order. In other words, the client does not need to worry about operations in its transaction being interleaved with those in another client's transaction. As long as each transaction leaves the database in a consistent state, the database will always remain in such a state regardless of crashes or concurrent access.

An object-oriented database is therefore more than a traditional database plus pointers and stored operations, because it adds also adds encapsulation, subtyping, automatic garbage collection, and so forth. Likewise, it is more than an object-oriented programming

language plus persistence, because of the safe, language-independent sharing inherited from its database lineage. This support for richly structured, persistent, and securely shareable data can allow OODBs to support new classes of distributed applications, a class that will grow in importance as computers being more ubiquitous, distributed, and interconnected.

2.2 Compromises of Existing Object-oriented Databases

Unfortunately, most object-oriented databases compromise the OODB ideal by failing to provide either safety or heterogeneous access in their client interfaces

Safety requires that client applications cannot violate object encapsulation, whether by fair means or foul, even when written in unsafe client languages such as C++ that allow unrestricted reads and writes to arbitrary memory locations. Many systems, especially commercial systems such as Gemstone [Butterworth 91] and O_2 [O Deux 91], sacrifice safety for performance by allowing clients direct access to object representations. (Databases expose representations in two ways: linking the client application directly into the database address space, or exporting copies of object representations into the client address space. Both options expose the clients to the danger we will describe here, although linking the client into the database is probably more dangerous.)

Such systems do not provide secure sharing since there is no way to guarantee that objects are used properly *(i.e.*, only by calling methods) while in user space. Languages such as C and C++ provide numerous loopholes that violate type safety and encapsulation, including pointer arithmetic, unrestricted typecasts, explicit deletion, untagged unions, *etc*.[1] When the object is returned to the database, its representation may have been replaced with garbage; the best the system can do is check that what comes back has the correct structure. (Surprisingly, despite the dangers, application programmers are willing to trust

[1] There are other, more subtle flaws in the C++ type system. Even without using explicit mechanisms such as typecasts, the application programmer can write code that is not typesafe. For example, C++ falls into a well-known pitfall by declaring that $S \subseteq T$ implies that $S[] \subseteq T[]$, where $\subseteq$ represents the subtype relationship and $S[]$ represents an array of S. This allows run-time type errors. For example, an *array of apples* could be passed as an *array of fruit*, since an apple is a fruit. But this allows an orange to be added to the array of apples, clearly a violation of type safety!

vital scientific and financial information to databases that make such compromises. Perhaps this is because they avoid sharing the database with other client applications, and believe that *they* would never write a badly behaved application.)

In addition, many object-oriented databases support only a single client language, usually a popular but unsafe language such as C++ (thereby *forcing* the application programmer to use an unsafe language), and therefore do not support heterogeneous access. Such databases (e.g., ObjectStore [Lamb 91]) sacrifice a key benefit of filesystems and databases: providing a means of exchange and communication between applications written in different languages. Regardless of the merits of any particular language, it is unrealistic to expect application programmers to restrict themselves to that language, because they may rely on the specialized features, libraries, or hardware support of some of other language.

2.3 Achieving Safety and Its Costs

These problems can be solved if client applications are isolated in their own protection domains. Protection domains improve safety because they can prevent client applications from violating object encapsulation. They also improve client language independence, because programs in some client languages do not easily co-exist with other programs in the same protection domain. For example, Lisp implementations typically assume they have their address space all to themselves.

This approach requires, however, that each time the application invokes an operation on an object in the database, a cross-domain call must be made. A significant fraction of database calls perform very little work (for example, looking up the value of an instance variable), so that the time to execute lightweight calls is dominated by the domain-crossing overhead. This makes the database prohibitively expensive to use for lightweight objects and operations.

Object-oriented databases are thus a prime example of the tension between safety and performance alluded to in the introduction. This thesis is concerned with techniques that it make it practical to keep domains separate by providing acceptable performance for lightweight cross-domain calls.

Chapter 3.

Thor Client Interface

Thor [11] is a new object-oriented database that aims to achieve the OODB goals mentioned in the previous chapter: safety, heterogeneity, and good performance, even for lightweight objects. Client applications written in any language can entrust persistent data to Thor, with the assurance that other applications or incorrectly implemented types that they do not use will not be able to corrupt that data. Our work was done in the context of Thor, so we will need to introduce the client interface supported by Thor. The remainder of this chapter is a complete description of the interface that Thor provides to clients.

3.1 Objects and Operations

From the client's standpoint, Thor provides a universe of encapsulated objects that can be referenced by client variables. Each object supports a set of operations. The client interacts with objects in the database by invoking operations on them. An operation can transform the internal state of its arguments and/or return one or more results to the client. (For example, the *deposit* operation on a bank account object might increase the amount of money in the account and return the new account balance to the caller.) Each operation is described by an interface that specifies the name of the operation, the arguments that it takes, the result that it returns, and other aspects of the operations behavior..

There are two kinds of results:

- *Handles* are opaque references to shared objects in the database. The object reference can be passed as an argument to other operations, but the object's representation cannot be accessed.

- *Values* are returned for the built-in simple types: integers, characters, reals, and booleans. Each of these has a defined external representation and corresponds to a built-in client language type.

Similarly, the arguments to operations are either handles, obtained as the result of previous calls, or values. The main point to realize is that by the client is not given the values for objects, except for a small, fixed class of objects. (Thor could support more complicated sorts of values (e.g., structures), but the issues raised in that case are orthogonal to this thesis.)

Handles are short-lived pointers that are valid only for the duration of a particular client session. They are actually represented by integer indices into a volatile array H in the Frontend that maps the handles to actual objects. Thor attempts to keep the range of handles as small as possible by allocating them sequentially.

When the client first begins a session, it is provided with a handle to the root of the database, which is a directory object that allows the client to look up the handles of other "well-known" objects via pathnames, similar to those supported by a file system. In general, however, the client finds objects by *navigation*— the client calls an operation that returns a reference to some other object, which in turn may be used to reach other objects. (Thor will also support queries, a specialized language for selecting some subset of objects based on the results of an operation applied to each object in a larger set [Hwang 94]. However, we do not discuss queries in this thesis.) Typically an application will do many stages of navigation before reaching an object where the values are of interest.

3.1.1 Iterators

In addition to procedures, Thor supports *iterators*. Iterators, introduced in the programming language CLU [Liskov 81], allow a Thor programmer to define new control structures specialized for a particular type of object. Rather than returning a single value, an iterator *yields* multiple values to the caller, which uses a special for construct to iterate over the yielded values one at a time. Each time the iterator yields a value, control transfers back to the caller, but the state and control flow of the iterator are maintained, so

that the iterator picks up where it left off to yield the value for the next iteration. (Essentially, the caller and iterator function as co-routines.)

For example, the following Theta code uses the nodes iterator to iterate over the nodes in a graph and mark each one.

```
for n: node in graph.nodes() do
    n.mark()
end
```

The code to visit all of the nodes in the graph may be complex and representation dependent; for example, it could perform a recursive a depth first search, a breadth first search using a queue, and so forth. But the iterator mechanism hides all of these details from the user of the type, so that even the most complex traversal mechanism appears to the client as a simple for loop.

3.1.2 Exceptions

In the normal case, Thor operations terminate by returning a value. However, Thor operations can also terminate by signaling an exception [Liskov 81, Goodenough 75], indicating that a condition has occurred that prevents the normal return value from being computed. An exception consist of a name and zero or more values that give additional information about the cause of the exception. For example, the array type might signal a bounds exception if the caller attempts to access a slot beyond the end of an array.

Each operation interface includes an explicit declaration of exceptions it might signal. In addition to the exceptions explicitly listed, *any* operation can signal the failure exception because of, for example, incorrect argument types provided by the client, concurrency control notifications that the current transaction will abort, *etc.*

3.2 Types

Thor programmers group objects in the database into *types* on the basis of behavior. Two objects of the same type share the same operation interface and semantics, even if their representations and operation implementations are entirely different. A type consists of a set of operation signatures, each of which includes the following:

```
% Calendar items
item = type
   % An item can be hilited in various ways.
   hilite_mode = oneof [ always, expire, never, holiday: null ]

   text() returns (string)
   set_text(t: string)

   dates() returns (dateset)
   set_dates(s: dateset)

   early_warning () returns (int)
   set_early_warning (days: int)

   hilite () returns (hilite_mode)
   set_hilite (mode: hilite_mode)

   equal (i: item) returns (bool)
end item
```

Figure 3-1: Example Type Interface

- the operation name
- the number and types of arguments expected
- an (informal) description of the operation's semantics
- the type of the return value.

Thor types are specified and implemented in the *Theta* programming language [Day 94], which was created in response to the shortcomings of existing languages with regard to safety and expressiveness. Client programmers can dynamically add new types and type implementations to Thor, but they must be written in Theta. The type interface describes the operations supported by the type but places no constraints on the possible implementations. Thor stores the interfaces for types in the database in a parsed form that is easy for applications (for example, an interactive browser) to read and interpret.

As an example, Figure 3-1 presents the interface for item type, which is used in Ical, a distributed appointment calendar written by Sanjay Ghemawat. An item in the calendar recurs over a set of days (represented by the dateset type), and can be highlighted in various styles.

3.2.1 The Subtype Hierarchy

A type system in which different types are entirely unrelated is too restrictive, because it fails to capture common behavior among types that are related but not identical. Thor therefore includes subtyping: a type *S* is a subtype of a type *T* if it supports the same[2] operations as *T*, plus possibly some additional operations. For purposes of typechecking, an object belonging to a subtype of a type T can be used as an argument anywhere an object of type T is expected.

For example, Figure 3-2 presents the *appointment* type, a subtype of the *item* type presented earlier. An appointment is a kind of item that supports the additional operations of getting and setting a specific time range for the appointment and maintaining a set of alarm times in advance of the actual appointment. An appointment can be passed to any

```
% Appointments are items that occupy a specified time range.
% They can also have associated times at which alarms can go off.
appointment = type item
    % ... all of the item operations are retained, plus the following

    % Appointment start (in minutes from midnight)
    start () returns (int)
    set_start (t: int)

    % Appointment length (in minutes)
    length () returns (int)
    set_length (l: int)

    % Optional alarm times (in minutes before beginning of appt)
    alarms () returns (sequence[int]) signals (no_special_alarms)
    set_alarms (list: sequence[int])

    % Specialized "copy"
    copy () returns (appointment)

  end appointment
```

Figure 3-2: Subtyping

[2] The signatures of the operations in the subtype *S* do not have to be exactly identical to those in *T*, but rather must *conform* to them according to the standard contra/covariance rules [Schaffer 85, Black 87, Cardelli 88]. For an operation in *S* to behave like the corresponding operation in *T*, it must be able to handle *at least* the same set of argument types, and must return a subset of the return types that the original operation would return. More formally, an operation $foo(s_1, s_2, ..., s_n)$ *returns* (r_s) in S conforms to an operation $foo(t_1, t_2, ..., t_n)$ *returns* (r_t) in T if, for each argument position *i*, s_i is a *supertype* of t_i and the return type r_s is a subtype of r_t.

function expecting an item.

3.2.2 Typechecking

As well as being useful for classifying and documenting object behavior for client programmers, types increase the safety of the system. Type information allows Thor to ensure that the client does not attempt to perform an operation on an object for which it was not intended (for example, hammering a screw). Without this information, clients might be able to corrupt objects in the database by passing them to inappropriate operations.

For client languages with a sufficiently rich type system, type errors can be detected at the time the client program is compiled. However, many languages allow an object of one type to be incorrectly forced to masquerade as another (incompatible) type, fooling the compile-time typechecker. Therefore, Thor typechecks client calls from any such language at runtime.

When the database receives a call, it verifies that the handles used are legitimate, that the object upon which the operation was invoked does in fact support the requested operation, and that the correct number and types of arguments were given. If the check fails, Thor does not perform the call. This ensures that operations always receive arguments of the expected types. If a client overwrites the storage used for handles, the worst it can do is make a valid call on an unexpected object, which must support an operation with the correct name and signature.

3.3 Transactions

Each operation call occurs as a part of an atomic transaction. The client commits the transaction with an explicit call, which implicitly begins a new transaction. Under the current system, there is only one active transaction at any given time; there are no nested or disconnected transactions. When the client attempts to commit the transaction, the system checks to ensure that the operations in the transaction would not be interleaved with those in some other transaction. If they would be interleaved, the transaction is

aborted, and the client must be prepared to redo the operations in the transaction. (See [Adya 94] for a more complete description of how transactions are implemented in Thor.)

3.4 Garbage Collection

To ensure safety, Thor performs garbage collection on the objects in the Thor universe. (If clients could explicitly delete objects, they might cause damage by deleting objects needed by other clients.) Handles are roots of the garbage collection: as long as the client references an object, Thor will not dispose of it. There is also a persistent root: all objects accessible from the persistent root are automatically persistent and are garbage-collected after they become unreachable. Both volatile and persistent objects are stored inside Thor; the only distinction between the two is reachability from the persistent root.

The fact that handles are returned to the clients is important with respect to garbage collection. If we returned permanent pointers to objects to the client we would not be able to garbage collect objects whose only reference was the pointer given to the client, even after the client session ended. The client could legitimately save a permanent pointer to a file, for example, and use it in a later session. Handles, in contrast, are only defined to be valid for the current session.

Because handles are roots for garbage collection, it is desirable that the database is notified when they aren't used anymore, even if the client session continues.

3.5 Discussion: Transmitting the Representations of Arbitrary Types

Our system uses handles to represent all database objects. Many other systems export the representation of database objects into the client address space, so that clients can access the objects without the expense of a cross-domain call. There are compelling reasons, however, why we do not allow this in Thor.

Most importantly, although it is possible to convert the representation of a Thor object into a corresponding client representation, there is no reasonable way to convert the object's methods, written in Theta, into methods in the native client language. Therefore the client program is forced to deal with the object representation directly. Without methods for interpreting them, the raw bits of the representation may be meaningless.

Also, a single type can have multiple implementations, each with a different representation. The client code will break when it receives an unexpected representation. For languages with very simple data models, such as Perl, even mirroring the representation may be a challenge.

Second, only immutable types can be transmitted in this way. If the client directly modifies the object representation, the database has no way of verifying that the changes maintain the representation invariant, so it cannot safely incorporate any changes back into the database. Even if the client were prevented from modifying a mutable object, it would not be possible for Thor to validate at commit time that all of the objects read by the transaction were up-to-date, because the object was not accessed using the standard method interface.

Finally, transmitting complex object representations would complicate the veneer and the database significantly (see [Birrel 94] for a description of a system with this ability.) Objects can contain references to other objects as well as data; the code to encode and decode the resulting object graphs can be time-consuming and a potential waste of bandwidth if the client is only interested in a small part of the data transmitted.

There are a few cases where we might relax our restrictions on transmitting representations. Sequences are immutable, ordered collections of objects. Because they are immutable and have semantics similar to arrays in most programming languages, a sequence of objects can safely and easily exported be into the client address space as an (immutable) array. (The database cannot prevent the client from modifying the array, but changes to the client array will under no circumstances be incorporated back into the database sequence, so it is effectively immutable.)

3.6 High-level Architecture

The objects of the Thor system are implemented in the *Theta* programming language [Day 94] and stored persistently on a set of distributed servers, possibly distinct from the client machines. Because there are multiple servers, accessing objects on the servers often involves network delay. To prevent network delays on every object access, a process

known as the *Frontend* caches objects and executes operations at the client node on behalf of the client. However, because Thor does not fully trust client applications, the objects reside in a different address space from the client program. (In Unix, the operating system on which Thor is implemented, protection domains, address spaces, and processes are all equivalent, so the client runs in its own process.)

The part of the system that allows the client to interact with Thor across the process boundary is known as the *veneer*, which is explained in detail in the next chapter. The corresponding part of the Frontend that receives requests from the veneer and invokes the appropriate operations in the database is known as the dispatcher. The dispatcher supports two interfaces: a textual interface, suitable for use by humans and veneers written in specialized text-processing languages such as Perl, and a binary interface, which provides higher performance and more direct encodings.

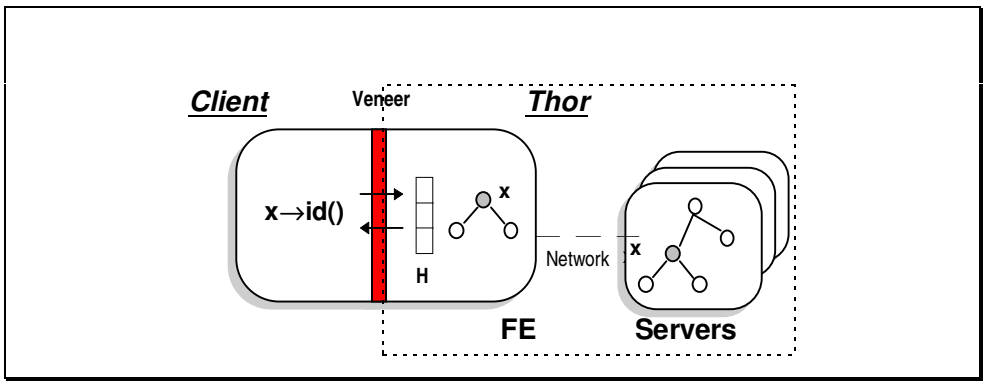

Figure 3-3: Thor Architecture

In Figure 3-3, the Frontend is caching a subset of the objects stored on the servers (including the object referenced by the client variable *x)*, and invoking the *id* operation on *x* on behalf of the client. The arrows in the diagram represent context switches to and from the database to perform the cross-domain call for the *id* operation. *x* is represented in the client address space as a handle, which is mapped by the handle table H in the FE to an actual object. When the client commits a transaction, any modifications to *x* and other objects cached at the Frontend are installed in the server once they have passed the concurrency control checks.

Batched futures are concerned with reducing the domain crossing overhead between the Frontend and the Veneer.

3.7 Safety issues

The Thor client interface ensures that the client cannot interfere with the correct operation of the database, even if the client language is unsafe or the veneer is not implemented correctly. (It is important that the database need not trust the veneer, since the veneer is linked into the client protection domain and could have its data structures corrupted by the client.) This claim of safety deserves close scrutiny. If it is false, the expense of placing the client in its own protection domain is wasted.

First, note that the client cannot directly violate encapsulation. The representations for the database objects are in a separate protection domain, so by definition the client cannot access them, assuming the protection domains supported by the operating system and hardware are worthy of the name. For the same reason, the client cannot jump to illegal points inside the database code or otherwise interrupt the database's normal flow of control. (Some microcomputer operating systems, such as Microsoft Windows, do not provide sufficiently strong protection domains. In that case, safety can be achieved by running the Frontend on a different machine from the client. In fact, if security were of paramount concern, we would do the same under Unix, since a user with root privileges or physical access to the machine can gain direct access to physical memory.)

What the client can modify are the handles stored in its own address space. However, no harm can come to the system as a result, because the system typechecks calls at runtime and only defines handles for objects that the client is allowed to access. If the client overwrites a handle with a value outside the legal range, or with a handle for an object of an incompatible type, the database will detect this fact when the client tries to use the handle in a call and abort the call with an error.

If the client overwrites a handle with a handle for another object of the same type, or some other type that happens to have the same interfaces for all of the operations the client uses, the database will not be able to detect the switch. However, all the client has succeeded in doing is redirecting a reference to another object that the client is allowed to access, which

in no way compromises the safety of the system. Notice the safety depends upon the fact that a handle is not assigned to an object unless the client has gained legitimate access to that object as the result of a method call. If every object had a preassigned handle, the client might be able to guess the handle of an object it was not allowed to access and forge a stub object with that handle.

Chapter 4.

Veneers

To use Thor from a particular language, that language needs a small extension called a veneer. The goal of the veneer is to map the features of the Thor client interface as naturally as possible onto native constructs in the client language. This chapter explains what the veneer does and how it is implemented. It assumes a simple model in which operations are actually performed at the point of the call and the client waits until they are complete; this assumption is relaxed in the next chapter.

The veneer consists of two major components:

- a small, fixed set of database commands for interacting with the database as a whole (e.g. starting a session, committing a transaction, etc.).
- a set of client types mirroring the types inside Thor. The client types implement *stub objects* that refer to objects in the database and *stub functions* that invoke the corresponding database operations. The net effect is to make Thor objects appear exactly like client objects.

For concreteness, we show first how these two components are implemented in the C++ veneer. We then consider the steps necessary to implement a veneer for a new client language, and show how the design changes depending on properties of the client language such as the richness of the type system and the availability of garbage collection. The implementation details of the veneer are primarily of interest to veneer implementers, and may be skipped without loss of continuity by those wishing only to understand batched futures.

4.1 Database Commands

The database commands allow the client application to make initial contact with Thor and to perform higher-level operations such as committing transactions. Only a small number

of general database commands are needed because most of Thor's functionality resides in individual objects and their associated operations, as discussed in the next section.

Figure 4-1 presents the C++ interface for the database operations; in every other client language they will appear essentially the same, modulo differences in syntax. The commands begin_session and lookup_wellknown allow a client to start a Thor session and obtain references to "well-known" objects by name. After invoking a series of object operations, a client can commit or abandon the work of the transaction by calling commit_trans or abort_trans, both of which implicitly begin a new transaction. Finally, end_session terminates a client session, after which point the client object references become invalid and the client can invoke no further operations.

```
/*
  Database Commands

  These functions are called directly by the client to initiate communication
  with the database, lookup objects, and define transactions.
*/

bool      begin_session(char *fe_spec);
//        requires -- fe_spec is a the name of a host, or NULL
//        effects -- attempts to open a connection with the specified frontend,
//        or with a newly create slave FE if fe_spec is NULL.  Returns
//        TRUE iff the connection attempt succeeded

th_any*   lookup_wellknown(char *wellknown);
//        effects -- returns the stub object named with the name wellknown, or
//        NULL if there is no such object.

bool      commit_trans();
//        effects-- Commits the current transactions and begins a new one.
//        Returns true if the commit succeeded.

void      abort_trans();
//        effects-- Aborts the current transaction.

bool      end_session();
//        effects-- Terminates the session with the database.
```

Figure 4-1: Database Commands

The implementation of the database commands is trivial, as seen in the example in Figure 4-2, because the database does all of the real work. The database commands simply send a cross-domain message to the database consisting of a command code and an encoding of

the arguments (if any), and read and decode the result sent back from the database, returning it to the client.

```
bool commit_trans()
{
    fputc('C', client_out);
    transfer_control_to_fe();
    return (getc(client_in) == SUCCESS_CODE);
}
```

Figure 4-2: Database Command Implementation

4.2 Stub Types

The more significant component of the C++ veneer is a stub type hierarchy mirroring the Thor type hierarchy. To use a Thor type, the client programmer includes the header file of the corresponding stub type and links in a library of compiled stub types when compiling the program. In the subsections that follow, we will first describe the stub types' interfaces, then their implementation.

4.2.1 Stub Type Interface

The C++ veneer header files declare classes corresponding to each of the Thor type interfaces. For example, Figure 4-2 presents the Thor interface to the Module type, and Figure 4-3 presents the corresponding C++ class interface, defined in the header file th_Module.h. (The Module type is used in the OO7 benchmark [Carey 92] and is intended to be typical of the sort of object used in computer aided design applications. The interface supports operations for returning a manual object that describes the module, finding the number of subassemblies, fetching a subassembly by index number, and getting the root subcomponent.)

```
Module = DesignObj type

    man () returns (Manual)
    numAssemblies () returns (int)
    assemblyIndex (i: int) returns (Assembly)
        signals (bounds)
    designRoot () returns(ComplexAssembly)



end Module
```

Figure 4-2: Type Interface in Theta

```
class th_Module : public th_DesignObj
{
    public:
        th_Manual* man();
        int numAssemblies();
        th_Assembly* assemblyIndex(int i)
            // signals(bounds)
        th_ComplexAssembly* designRoot();
        free ();
    protected:
        th_Module(handle h) : th_DesignObj(h) {}
};
```

Figure 4-3: Stub Type Interface in C++

The C++ interface is essentially a direct translation of the Theta interface into the C++ syntax. Because the interface preserves the subtype relationship declared between Module and DesignObj, the C++ compiler is able to do compile-time typechecking for stub type operations. There is one substantial difference: the way that exceptions are handled. This mapping of exceptions into C++ is explained in full in Section 4.2.1.2. There are also several cosmetic differences between the two interfaces:

- Thor type names are prefixed by th_ to reduce name clashes with client types
- The implicit pointers in the Theta interface are made explicit, following C++ conventions. (This is also necessary because restrictions in C++ on the way types are used. If type S declares a variable of type T and vice versa (for example, in an argument declaration), then it is necessary to use pointers for the declarations, because C++ must be able to determine the size of any variable at the time a type is declared.)
- The interface includes a free operation for deallocating the storage associated with the stub object and informing the database the object is no longer referenced by the client program. (We will discuss storage management in greater detail in Section 4.2.3.) The interface also includes a private constructor for use by the veneer. It cannot be called by the client, but is used by the veneer to construct a new th_Module object given a Thor object handle.

4.2.1.1 Example

Before going into the details of the veneer, we will consider a brief example to show how the veneer is actually used. The traverse_part client function does a depth-first traversal of all of a graph of atomic parts in a OO7 database. The th_AtomicPart type is a stub type define by the veneer, while the IDSet type used to keep track of the parts already visited is implemented entirely in the client language. The only point we want to emphasize is that the two types are used identically by the client programmer; the veneer makes the indirection to the database invisible.

```
#include "th_AtomicPart.h"
#include "th_Connection.h"
....
int traverse_part(th_AtomicPart *a, IDSet *visited)
{
    int id  = a→id();
    if  (visited→contains(id)) return 0;   // don't visit same part twice

    visited→insert(id);                    // Add part to list of visited parts
    int result = doPart(a, id);            // Perform action on part
    int num = a→numOutgoing();
    for (int i=0; i < num; i++) {          // Traverse the subparts
        th_Connection *next = a→outgoingIndex(i);
        result += traverse(next→to(), visited);
        next→free ();
    }
    a→free();
    return result;
}
```

Figure 4-4: Example Client Routine

4.2.1.2 Handling Exceptions

Exceptions are officially a part of C++, but many compiler implementations do not support them. Other popular languages do not support exceptions at all. For these reasons (among others[3]) our C++ veneer does not use exceptions directly.

In performing this mapping of exceptions to C++ without using an exception mechanism, we want to maintain as many of the desirable features of exceptions as possible. In particular, features we would like to retain are:

---

[3] Another reason is that C++ exceptions are not compatible without futures, whereas the technique to be described is.

- The client should not have to check for exceptions after every operation, since this interrupts the flow of code and makes it hard to read.
- However, exceptions should not go entirely undetected by the client.
- If the client does fail to handle an exception, the veneer should be able to detect this fact.

In addition, we want the mappings of exceptions into the client language to be as unobtrusive as possible in terms of its effect on stub function signatures. For example, we would not want exceptions to require an additional argument for every operation, since this is inconvenient for the client programmer and reduces the similarity between the veneer and database type signatures.

Our solution has two major features: an exception history that allows the client to defer checking for exceptions, and exception propagation, which allows the client to determine if exceptions occurred for any intermediate calls in string of interrelated calls by checking only the final result.

The exception history is maintained by the veneer and contains a history of the exception results of all "recent" calls, where a recent call is a call in the current transaction. If no exception is signaled by an operation, the veneer returns the normal return value and adds a null exception value to the history, indicating that no exception has occurred. If, on the other hand, an exception is signaled, the veneer returns a null value for the result (that is, an invalid handle for stub objects, 0 for integers, etc.) and adds an exception value to the history giving the name and associated values for the exception. At a later point, the client can obtain the exception value associated with a call and, by invoking methods on the exception value, determine the properties of the exception result.

Before proceeding to a description of how the client obtains exception values, we will first consider exception propagation, which is possible for operations that return handles but not for those that return basic values. When an operation that returns a handle signals an exception, that stub object is tagged as invalid. If the client uses the handle as an argument to another call, the veneer detects this fact and returns another exception value

with the name unhandled_exc and a value consisting of a pointer to the exception value that the client originally failed to check. Continuing the propagation, if a handle associated with an unhandled_exc is passed as an argument, another unhandled_exc exception is returned that points to the *original* exception value. Thus, if an exception is signaled anywhere in a string of interrelated calls that return handles, the exception will propagate all the way to the final call. To determine what exception occurred originally, the client need only check the final exception value. Exception propagation is not possible for calls that return basic values because in general the entire range of a basic value is legal as a return value.

We now return to a discussion of the client calls for obtaining exception values. The exception history provides several different ways for the client to obtain exception values, each providing different tradeoffs with respect to the unobtrusiveness and precision of the exception handling. The last_exc() call returns the exception value for the most recent call. The exc() method defined on stub objects returns the exception value for the call that returned the stub object. Finally, the first_exc() call, followed by repeated call to next_exc(), allows the client to iterate through all of the unhandled, non-null exceptions in the history. We expand on each of these calls in the paragraphs below.

The last_exc() call supports a style of exception handling in which the client checks for exceptions immediately after each call. This style of exception handling is inconvenient, in that it tends to break up the flow of the client program, but is necessary if exceptions are to be detected immediately. In most cases, the client will need to call last_exc() after every call that returns a basic value to make sure the result is valid.

The exc() call allows a more convenient way to handle exceptions for calls that return stub functions. The client can defer checking for exceptions until any later point, because as long as the client holds onto a stub object, it can retrieve the associated exception value. Because exceptions propagate for calls that return handles, the client can even avoid checking for exceptions from most calls. If the client needs to find the precise exception that set of a chain of exceptions, the client can obtain it by getting the value associated with the unhandled_exc error.

Finally, examining the exception history provides an "optimistic" exception handling mechanism. If the programmer believes that exceptions are unlikely, they can optimistically assume that none will be signaled and defer checking for exceptions until commit time. At commit time, the client program can verify that there are no items in the unhandled exception history. If there are exceptions in the history, they can abort the transaction, undoing all of the work in the transaction, and start again, or they can attempt to unwind its state back to the point of the exception, although this is probably quite difficult. For this approach to work, the client has to be prepared for possibly meaningless null values from basic valued calls that signal unchecked exceptions, even if the operation specifications say that such values are not returned.

The exception history does more than allow the client to handle past exceptions; it also allows the veneer to determine when the client has failed to handle an exception. Each non-null exception value in the history has a flag, initially false, indicating whether it has been checked yet by the client. When the client obtains an exception value using one of the calls mentioned above, the checked flag is set to true.

At commit time, the veneer verifies that every non-null exception value in the history has been checked; if not, the veneer refuses to commit the transaction. The intention is that the client should be aware of any exceptions before deciding that the work of the transaction should be committed. The client can override this is on a per-transaction basis by calling exceptions_clear() immediately before committing. Exceptions are automatically cleared out when the client aborts a transaction and after a transaction commits successfully.

In practice, clients will probably use a mixture of these exception handling styles, depending upon the nature of the exception. The exceptions explicitly listed in the type interfaces will usually need to be checked for immediately (for example, the bounds exception for an array.) However, many clients will probably defer checking for the failure exceptions until the end of a loop or even until the end of the transaction. These exceptions are often not caused directly by the client and will not necessarily reoccur if the transaction is restarted, so it makes sense for the client to simply abort and try again.

4.2.1.3 Exception-Handling Example

Consider a withdraw operation for a bank account that signals two exceptions: overdrawn and not_possible. Each exception has associated information: an integer for overdrawn, giving the amount the account is overdrawn, and a string for not_possible, giving an English explanation for why the caller is going to go penniless.

The Theta interface for withdraw is:

```
withdraw(dollars: int) signals (overdrawn(int), not_possible(string))
```

The corresponding C++ stub function interface is:

```
void withdraw(int dollars);  /* signals (overdrawn(int), not_possible(string)) */
```

Figure 4-5 shows how a C++ application would handle the exceptions signaled by withdraw, an operation that does not return any values. The exception value supports methods for each of the exceptions that the corresponding operation might signal; the methods return true if the exception with that name has been signaled and false otherwise. In addition, the exception value supports methods for each value associated with an exception name; for example, overdrawn_int is used in the figure is used to retrieve the integer value associated with the overdrawn exception. The client program uses these methods to check for exceptions and retrieve their values.

```
my_account→withdraw(100);
if (last_exc()→overdrawn())                    // check for exceptions
    cout << "Your account is overdrawn by " << e→overdrawn_int() << " dollars";
else if (last_exc()→not_possible())
    cout << "Withdrawal is not possible because: " << e→not_possible_string();
```

Figure 4-5: Basic Exception Handling

Similarly, Figure 4-6 shows how the client would check for exceptions from the next operation on a list, which returns a stub object and signals the bounds exception. The only real difference between the two cases is that the exception value is obtained by invoking exc() on the result object; this would work even if other operations had been invoked in the meantime.

```
l = l→next();        // get the next item in a list
if (l→exc()→bounds()) // check for the bounds exception
    ....
```

Figure 4-6: Stub object Exception Handling

4.2.2 Stub Type Implementation

The previous subsection showed how the veneer maps the interfaces of Thor types onto corresponding client interfaces. This subsection is concerned with how the veneer exports the functionality of Thor objects without compromising their safety or changing their semantics.

4.2.2.1 Basic Values

Basic values (integer, characters, reals, and booleans) are represented as native client values, transmitted using a suitable representation to and from the database and decoded by the veneer. (For example, the num_assembly stub function returns a native C++ int.) In our system, basic values have the same representation in Thor and in C++, so the encoding and decoding steps are trivial; the representations can be copied directly to and from the communications buffers. In other languages, the decoding steps might be more substantial. In any case the burden of encoding and decoding should fall on the veneer, not the database. Doing encoding and decoding in the database would require modifications to the database for each new language, whereas doing them in the veneer offers opportunities for improving performance because they can be done lazily, as we shall later see.

4.2.2.2 Object References

Thor objects, in contrast to basic values, are represented by reference, not by translations of their actual representations. Although the stub type interfaces encourage client programmers to believe that they are pointing directly to Thor objects, the actual Thor objects are safely stashed away in a separate address space. What the client is actually pointing to are *stub objects*.

The representation for every stub object is the same: a single, protected field containing the handle for the corresponding Thor object. (By protected, we mean that the handle field can be accessed only within the th_any class and its subclasses, not by the client.) The stub objects thus encapsulate a reference to a Thor object without revealing its actual representation, even if the client violates encapsulation to access their contents. This scheme is depicted in Figure 4-7.

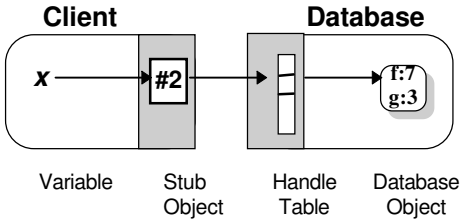

Figure 4-7: Object Representation

There is no legitimate way for the client to access the handle stored inside of the stub object, make a copy of a stub object, or create a new stub object without invoking a Thor operation. Although clients may do these things illegally by violating encapsulation, they can only cause trouble for themselves by doing so (especially with the optimizations discussed later) and cannot cause damage to the objects in the system.

Stub objects are allocated on the heap when Thor returns a handle as a result of an operation. To ensure that exactly one stub object is allocated per handle, the veneer keeps a table VH that maps between handles and stub objects. (VH is analogous to the table H in the database that maps handles to actual objects.) When the veneer receives a handle from Thor, it checks the corresponding slot in VH to see if a stub object has already been allocated, returning a reference to that object if so. If not, the veneer allocates a new stub object and stores a pointer to it in VH for later use.

4.2.2.3 Stub Functions

A stub function implements a Thor operation by making a cross-domain call to the database, which performs the actual operation and sends back the result. More specifically, the stub function marshals the function name and arguments into a message to the server and transfers control to the database protection domain, as depicted in Figure 4-8. The database dispatcher receives the interprocess message, looks up the corresponding

method and objects, typechecks the call, performs it, and sends the results back to the client in an interprocess message. Finally, the client stub receives the results and returns it to the client program.

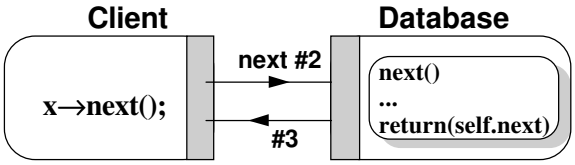

Figure 4-8: Cross-Domain Method Calls

Thor stub functions are similar to those used in standard RPC [Birrel 83], with two differences:

- all objects are passed by reference (using handles), not by value, so that the encoding for objects is much simpler.
- The database performs a method dispatch, not merely a function call. The actual code that is invoked depends upon the receiver's type as well as the method name. (Other RPC systems (for example, [20]) also provide this facility.)

As an example, Figure 4-9 gives the implementation for the concat stub function defined on mstrs (mutable strings.)

```
th_mstr* th_mstr::concat(th_string* s)
{
    th_mstr* res=0;
    begin_invoke(self, 2);
    put_handle(s->handle);
    if (do_invoke()) {
        NEW_TH_OBJ(get_handle(), th_mstr, res);
    }
    else {
        get_exception();
    }
    end_invoke();
    return res;
}
```

Figure 4-9: Example Stub Function

Stub functions have a simple, fixed structure, with each line in the stub sending some part of a call to the database dispatcher. The stub function first sends the receiver and method index using begin_invoke, followed by each of the arguments. do_invoke() transfers

control to the database and returns TRUE if the operation succeeded. If do_invoke() indicates success, the stub creates a new stub object of the appropriate result type, initialized with the result handle read from the database. If not, the else branch contains a call to read the exception value. The stub then calls end_invoke() and returns the result object to the client. (It may seem that end_invoke() is superfluous. Indeed, for normal invocations, end_invoke() does nothing. However, by changing the behavior of end_invoke() and get_handle(), the veneer can dynamically switch from mode without futures to one with futures, and vice versa. Every other stub function follows exactly the same pattern; as we shall see, this makes stub functions easy to generate automatically based on their interfaces.

4.2.3 Storage Management

Some client applications can quite happily ignore the issue of deallocating stub objects. The individual stub objects are compact; as long as the client accesses a reasonable number of distinct objects the total space taken up by stub objects will not be too large, and as soon as the client application terminates the database can reclaim the handle table space and garbage collect the objects whose only references were the handles given to the client.

However, in general, the veneer needs some of way of knowing when a handle is no longer needed so that it and the database can reclaim storage associated with the handles. For example, some applications may access very large numbers of objects (such as traversals that visit every item in the database), or create many temporary objects that are only used for a short time.

The current C++ veneer implementation requires that client applications explicitly free stub objects by calling the free method on them. When the client frees a stub object, the client must no longer use any other references to it. This is a reasonable requirement, given the C++ philosophy, but what the client cannot be expected to know is when two calls happen to have returned the same object. To behave reasonably in this case, the veneer handle table VH includes a counter for each handle *h*, incremented whenever a stub function returns *h* to the client and decremented each time the client calls free_object on

h. Only when the count reaches zero does the veneer reclaim the slot in VH and send a message to the database saying that *h* is no longer in use.

4.2.3.1 Smart Pointers

C++ programmers are accustomed to explicitly deallocating dynamically allocated objects, so our convention of using a free operation will not seem alien to them. However, it is possible for the veneer to lift this burden from programmers by defining "smart pointers" that automatically maintain a reference count for each stub object [Stroustrup 87]. Rather than accepting and returning normal pointers, the veneer operations are redefined to use smart pointers. When the reference count for a stub object drops to zero, that stub object is automatically destroyed. Because the client program references stub objects exclusively through smart pointers, it never needs to worry about freeing them.

Like a normal pointer, a smart pointer supports the * and → dereferencing operators to return the referenced object. In addition, however, the smart pointer type overloads its assignment, copy, and destructor operations to update the reference counts of the associated stub objects. Figure 4-10 defines a smart pointer to a th_any, assuming that th_any has a reference count field refs initialized to zero.

Smart pointers and other reference counting schemes typically have the problem that they cannot garbage collect cycles of references. For example, if A points to B and B points to A, their reference counts will both be at least 1 and they will never be garbage collected. Fortunately, this situation does not happen with stub objects. Stub objects only "point" (via handles) to objects in the database, and database objects do not point to stub objects, therefore a cycle including a stub object is impossible, so every stub object that can safely be reclaimed will be.

One complication of using smart pointers with subtyping is that is necessary to mirror the subtyping relationships in a hierarchy of smart pointer types, rather than creating a single parameterized smart pointer type. Although C++ supports parameterized types, there is no subtyping relationship between different instances of a parameterized type, so a parameterized smart pointer class does not allow the Thor subtype relationships to be

captured. For example, if S is a subtype of T, then there is no subtyping relationship between Ptr<S> and Ptr<T>, where Ptr is a smart pointer type. Instead, the smart pointer implementer must create Ptr_S and Ptr_T smart pointer types, with Ptr_S explicitly declared to be a subtype of Ptr_T.

```
class smart_ptr {
    public:
        th_any *obj;
//   Increment and decrement the reference counts of the object
        void    inc_refs() const   {if (obj) ++obj->refs;}
        void    dec_refs const     {if (obj) {if (--obj->refc < 0) delete obj;}
// Unless the pointer is null, increment the refs when a
// smart pointer is created or copied
        smart_ptr() : obj(0) {}
        smart_ptr(th_any *t) : obj(t) {inc_refs();}
        smart_ptr(const smart_ptr& o) : obj(o.obj) {o.inc_refs();}
// If smart pointer b is assigned to a, increment b's refs and
// decrement a's
        smart_ptr& operator =(const smart_ptr& o)  {
            o.inc_refs();        dec_refs();
            obj = o.obj;
            return *this;
        }
// Decrement the reference count when a smart ptr is destroyed
    ~smart_ptr()  { dec_refs();}
// Define operators to convert a smart pointer into an actual pointer
    operator smart_ptr*()  {return obj;}
    th_any* operator->() {return obj;}
}
```

Figure 4-10: Smart Pointer Implementation

We implemented smart pointers for our C++ veneer. Unfortunately, our C++ compiler adhered to an obsolete version of the still evolving C++ standard that allowed extremely aggressive deletion of the compiler temporaries used to store smart pointers, leading to the premature destruction of stub objects. At least for now we are using the free approach instead.

4.2.4 Veneer Support For Iterators

There are two issues associated with supporting iterators in client programs. First, most client languages do not directly support iterators. In addition, on the database side, it is expensive to maintain the state associated with the iterator between calls. Implementing yield directly would require that the database use a separate thread for each iterator called

by the client; the thread would be awakened when the client requests the next value from the iterator.

To avoid this complexity, the veneer presents an iterator to a client as an operation that returns an array that contains as elements all of the values yielded by the iterator. (Note that this does not change the implementation of iterators inside the database; it's simply a way of packaging the iterator construct in a way that can be used from the client language.)

For example, suppose a CompositePart object had a subparts iterator that yielded its constituent AtomicParts:

```
CompositePart = type
    subparts() yields (AtomicPart)
    ...
end CompositePart
```

In the C++ veneer, the iterator would be mapped to the function:

```
Array<th_AtomicPart*> subparts_array()
```

where Array<th_AtomicPart*> is a parameterized array type, instantiated to store references to th_AtomicParts. If the iterator signals an exception at any point, iteration stops and the correspond slot in the array is set to a null value; the exception value can be determined by calling last_exc.

To iterate through the AtomicParts array returned by subparts, a C++ client programmer could write:

```
Array<th_AtomicPart*> ap = cp→subparts_array()
for(int i=0; i < ap→size(); i++) {
    th_AtomicPart* p = ap[i];
    ...
}
```

Bundling the results of an iterator as an array fails to provide one important feature of iterators: early termination. The client has no way of stopping the iterator before it has generated all of its values. This can be a problem if the iterator generates a large number

of values that the client is not interested in, or the iterator has side effects and must not be allowed to run longer than the desired number of times. Also, some iterators generate an unbounded number of values, for example an iterator that generates all of the prime numbers.

The problem can be partially solved by allowing the client to specify how many times the iterator is allowed to run in advance. However, this approach is of no use if the number of iterations cannot be determined in advance. For example, with the prime number iterator mentioned earlier, the caller could find the 10,000th prime, but not the largest prime less than 10,000. In the Batched Control Structures section (Section 7.2), we describe a scheme that does allow early termination of iterators, and that provides (with some restrictions) the additional benefit of allowing the entire body of the foreach loop to run inside the database without domain crossing or typechecking overheads.

4.3 Implementing New Veneers

We will now consider the steps required to implement a veneer for a new language. Much of a new veneer can be copied from an existing veneer. There are only a handful of database commands, and it is trivial to reimplement them by hand, so we won't discuss them further. This leaves the stub types: as we shall see, implementing the stub types is mostly a matter of tweaking an existing *stub generator* to output stubs following the syntax of the new language. However, we also consider how the veneer design changes in response to some significant differences between client languages, such as varying type systems and garbage collection.

4.3.1 Stub Generators

In principle, the veneer implementer might implement all of the stub type interfaces and implementations by hand, and in fact this has been done for small subsets of Thor. For the complete Thor type hierarchy, however, this is too much work, especially since the Thor type hierarchy is constantly growing as new types are added. (There are currently several dozen types and hundreds of methods.) Therefore, the veneer implementer instead writes a *stub generator*, which automatically generates the stub type interfaces and

implementations given the interfaces to the Thor types. This is essentially similar to the stub generator used in RPC [Birrel 83].

The stub generator runs as a part of the database and has a fairly simple implementation. Given a method interface for a type T of the form:

method (*arg1*: *T1, arg2*: *T2, ...*) returns (*result*)

the C++ stub generator constructs a stub function based on the skeleton in Figure 4-11 with each of underlined items replaced by the corresponding item from the method interface. Because the database stores a parsed version of the method interface, it is easy for the veneer generator to obtain the appropriate items with which to "fill in the blanks."

```
result * method (T1 arg1, T1 arg2, ...)
{
    result* res=0;
    begin_invoke(self, method-index);
    put_handle(arg1); put_handle(arg2); ...

    if (do_invoke())
        NEW_TH_OBJ(get_handle(), result, res);

    end_invoke();
    return res;
}
```

Figure 4-11: C++ Stub Function Skeleton

(For explanatory purposed, we have simplified the skeleton somewhat. For example, the call used to send an argument is not always put_handle but rather depends upon the type of the argument being sent.)

Constructing a new stub generator is essentially a matter of modifying an existing stub generator to use the syntax of the new language. For example, Figure 4-12 shows the skeleton that would be used for an Emacs Lisp stub function. It conceptually the same as the C++ version, though surface differences show up because of Lisp's different syntax and lack of explicit type declarations. (Notice that the method name is prefaced with the name of the type to ensure uniqueness, and that the implicit *self* argument in the Thor version is made explicit. In the next section, we will consider in more detail these and other changes caused by differences in the client language.)

```
(defun T-method (self arg1  arg2 ...)
    (begin-invoke self method-index name)
    (put-handle arg1)  (put-handle arg2)  ...
    (let (result  ((if (do-invoke)
                       (get-handle) nil))
        (end-invoke)
        result))
```

Figure 4-12: Lisp Stub Function Skeleton

Currently, the stub generator is implemented as a program running in the database that is explicitly modified to support new languages; this has proved reasonably convenient. One might build a "universal" stub generator, which takes a description of the stub function skeleton for an arbitrary language and automatically generates the stub functions directly from the description. The challenge is to make skeleton description language sufficiently powerful without making it as difficult to use a full-fledged programming language. (For example, some form of looping is required, as implied by the ellipses in the skeleton examples.) This might be an interesting direction for future work.

4.3.2 Different Type Systems

The C++ type and function dispatch system are rich enough to completely mirror those of Thor. (In particular, the C++ type system supports subtyping, and the dispatch mechanism allows the same method name to dispatch to different functions, with the appropriate function chosen based on the type of the receiver.) This allows the C++ compiler to typecheck Thor calls at compile time , and also permits method names to be used "as is". In this section, we consider how the veneer interface changes for languages with more restrictive type and function dispatching systems.

4.3.2.1 Weakly-typed Languages

Languages such as Lisp have the simplest possible type system, in which there are no explicit type declarations and function dispatching depends only on the name of the function, not the type of the receiver. As we saw in Figure 4-12, this requires several changes to the stub function interfaces. First, the method name is prefaced with the type name, to ensure uniqueness; for example, the next operation of the list type would be called list-next in the Lisp veneer. (The type and method name are separated by a '-', a

character that never occurs in Thor type or method names, so uniqueness of the stub function name is assured.) Second, the implicit self parameter in the Theta version is made explicit. Finally, the type declarations for the arguments are completely omitted in the actual code, though they should be maintained in a comment for the benefit of the client programmer. Any type errors in a weakly typed language will of course go unreported at compile-time, but Thor will detect and report any type errors at run-time.

4.3.2.2 Type Systems without Subtyping

Many older statically-typed languages, such as C and CLU, support explicitly declared types, but do not have any form of subtyping or method dispatching. The name clash problems caused by the lack of method dispatching can be solved in the same fashion as for the weakly typed language (by concatenating the type and function name); indeed this is style already required by CLU and commonly used by C programmers.

The lack of subtyping is more of a challenge. The cleanest approach is for each type to include explicit conversion functions that convert between it and its immediate supertypes, thus giving the client the ability to use the subtype wherever the supertype is expected, without allowing any illegal conversions. (Because every stub object has exactly the same representation, it is trivial to convert a stub object of one type into a stub object of another type.) For example, if type D has supertypes B and C, which have supertype A, then stub type D will include conversion functions to B and C, which will include conversion functions to A. To make things more convenient, each subtype includes all of the methods of its supertypes with the receiver type declaration specialized to the subtype, so that no subtype-to-supertype conversions are ever necessary for the receiver argument.

A less clean, but more convenient approach is currently used in our C veneer, which treats C as if it were a weakly typed language. Each of the stub types is given a different name to serve as documentation, but they are all typedef'd to the same type for purposes of typechecking. This eliminates the need for explicit subtype-to-supertype conversions but also eliminates any possibility of compile-time typechecking.

4.3.2.3 Type Systems without Multiple Supertypes

Finally, languages such as Modula-3 and Object Pascal have subtyping and dispatching, but do not allow a subtype to have multiple supertypes. Our solution is basically the same as in the previous section: a type is declared to be the subtype of any one of its supertypes (chosen arbitrarily) but includes conversion operations to convert it to any of the other supertypes. Few explicit conversion are likely to be required because the client language already does the right thing for single supertypes, and because multiple supertypes are currently not used very frequently in Thor.

4.3.3 Coordinating Client and Database Garbage Collection

Some languages, such as Lisp, Smalltalk, and CLU, have a garbage collector that automatically disposes of objects that are no longer referenced. Thus, garbage collection performs a function similar to that performed by the smart pointers defined as a part of the C++ veneer. However, because the garbage collector is a built-in and often non-customizable part of the system, it is difficult to achieve the necessary degree of cooperation between the veneer and database garbage collectors, raising issues that we discuss in this section.

When a stub object is disposed of, the database needs to be informed so that it knows that the database can reclaim the handle table slot and potentially garbage collect the object. Some garbage collection systems, such as that of Smalltalk, allow user-defined "finalization hooks" that are automatically run when an object is about to be disposed of. In such a language, the hook can easily be defined to inform the database of the stub object's impending demise. Other languages, such as Lisp, do not provide such hooks. In that case, client programmers must explicitly call a free operation to release handles and allow the database to garbage collect the associated objects. If such an operation is not provided, the stub objects in the veneer will be automatically garbage collected, but newly created or newly unreachable objects in the database will not be subject to garbage collection until the client program terminates.

A second issue is how to avoid creating redundant stub objects while still allowing stub objects to be garbage collected. Recall that we used a table VH to map between handles

and existing stub objects, which we used to avoid creating a new stub object if the same handle was returned again by the database. In a system with garbage collection, the references in VH will prevent the stub objects from ever being garbage collected!

Smalltalk provides a special mechanism for avoiding this problem: weak arrays. A weak array is like a standard array, except that references in it are not traced for purposes of garbage collection, thus allowing the referenced items to be potentially garbage collected. Furthermore, the entry in the array is automatically cleared when the associated item is garbage collected, allowing the veneer to detect whether the associated stub item still exists.

In the absence of weak pointers or arrays, there are two not entirely satisfactory compromises possible:

- allow redundant stub objects. This will cause additional work for the garbage collector, and will waste storage if the client maintains references to many redundant stub objects.
- prevent stub objects from being garbage collected unless the client explicitly clears out the entry in VH by calling free on the object. The disadvantages are the dangers and inconveniences of explicit deallocation. (If the stub object has other references in the client, but the corresponding Thor object has no other references, the Thor reference could be garbage collected, leaving the veneer references dangling.)

The general problem is that the veneer and the database need to perform a simple form of distributed garbage collection, the need for which was not anticipated by many language designers. Unfortunately, unless the garbage collector provides sufficient hooks for customization, we don't have any magic bullets for remedying these shortcomings.

4.4 Language Independence Issues

We claimed that the veneer design was language independent. We will argue that it is not only possible but also *easy* to implement the veneer design in essentially any client language. Several features of the design contribute to this property. First, the object

representation used is quite simple. The client language only needs to be able to represent integer handles, plus a fixed set of basic types.

Second, generating the stub functions for an arbitrary client language is easy. The veneer generator itself is fairly simple, because each stub function has exactly the same structure, and because the database stores the interfaces of the types it supports in an already-parsed form. Adding support for a new language is as simple as modifying an existing veneer generator to produce output in the syntax of the new client languages. For example, we were able to quickly modify the C stub generator to fully support C++, even though the class and subtyping mechanisms in C++ are much richer than what is available in C. Similarly, [Helfinstine 94] describes his implementation of a Modula-3 veneer for Thor; the author reports that it was not difficult to modify the C++ stub generator to output stubs for Modula-3.

The only non-trivial requirement to implement a veneer is that the client language support a cross-domain communication mechanism (for example, interprocess communication) so that it can exchange message with the databases. Most client languages have such support or allow it to be added in an extension language. Attesting to the portability of the design, we have implemented veneers for C, C++, Emacs Lisp, Perl, and TCL.

Chapter 5.

Batched Futures

This section discusses batched futures, the main contribution of this thesis. It begins with an abstract description of futures and how they are used to achieve transparent batching. It then presents an efficient implementation of batched futures in Thor.

5.1 Introduction to Futures

In general, a future can be viewed as a reference to the eventual result of a call. Like the actual result, it can be passed as an argument to other calls, included in data structures, and so forth. If the actual value referred to by the future is required and not yet computed, the client waits until it is available. Futures were introduced in the parallel programming language Multilisp to hide caller-callee parallelism, but we use them for a different reason: to defer calls so that they can be batched.

An important point to realize is that for many calls, the actual return value is not of immediate interest to the client. This is especially true in a system such as ours that conceals the representation of objects through handles: the particular value of a handle is *never* of any interest, since the only thing a client can do with a returned handle is pass it as an argument to another call. (In many veneers, client programmers will not even know that handles exist because they are encapsulated inside of stub objects.)

The basic idea of our batched futures design, in its simplest form, is to batch calls as long as they return handles. When the client makes such a call, the stub function records information about the call, and returns a *future* to the client instead. In our system, the future is essentially just an integer chosen by the *veneer* to stand for the eventual result of a particular call[4]. Later calls in a batch can refer to results of earlier ones using futures

[4]Similarly, a handle is really just a arbitrary integer chosen by the database to designate a particular object; there is no particular significance to the actual value.

and thus a sequence of interrelated calls can be batched together. As soon as the client makes a call that will return a basic value, or commits a transaction, the veneer sends the entire batch of calls to the database in a single domain crossing. As the database processes each call, it makes a mapping between the result and the corresponding future to allow the result to be retrieved for later uses of the future, and sends back to the veneer the actual values for each of the futures in batch.

Note that the mechanism depends upon knowing the signatures of the database methods, so that the veneer can tell whether to send a call immediately or defer it. This information is available in the interfaces stored in Thor and embedded in the stub functions stored in Thor. Stubs that return handles defer their calls for later execution; other stubs cause the execution of their call and the preceding batch of deferred calls.

5.2 Example

Consider the *nth* function, which a client application might define to return the nth value in a list of integers as seen in the following pseudocode:

```
nth(l :int_list, n: int) returns(int)
    for i = 1 to n do
        l = l.next()
    return l.first()
end nth
```

It works by calling the next operation *n* times to find the *n*th node in the list. Then it uses first to get the integer associated with that node. Using standard invocations, the function requires *n*+1 pairs of domain crossings.

With batched futures, the same code requires only a single pair of domain crossings. Inside the for loop, the client code makes *no* domain crossings. The next stub function in the veneer simply returns futures $f_1..f_n$ to stand for the results of the calls and adds information about each call to a queue of batched calls. Each future is used as the receiver (i.e., first argument) of the following call. Finally, the client calls l→first(), which returns a basic value. The stub function for first sends the batch of calls to the database, which

evaluates the set of batched calls and sends back the requested result for first. This entire process is depicted in Figure 5-1.

In essence, the batched calls form a simple program that recreates the effects of a set of possibly interrelated calls. The programming language allows just a few actions:

- calling an operation
- assigning the result of an operation to a future usable as an argument to later calls
- returning the result to the client

The database's job is simply to interpret this programming language efficiently. We will describe how it does so in the next section.

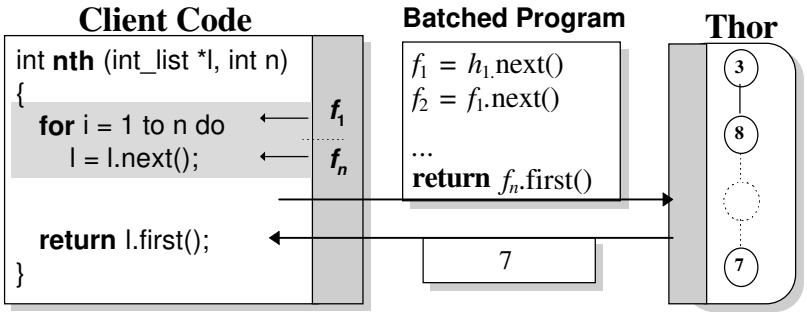

Figure 5-1: Batching interrelated calls using futures

The *n*th function is not necessarily a realistic example, because it would logically be included as operation in the database. However, one aspect of the example is realistic: it navigates a chain of pointers before reaching a point where actual values are of interest. Clients often navigate other, less predictable chains of pointers. It is not realistic to expect that the database includes a built-in operation for every chain of pointers that the client might follow, both because it is difficult to anticipate every useful chain and because to do so would create a very cluttered interface. Batched futures help resolve the dilemma: type interfaces can consist of a logical set of operations, possibly fine-grained, that are combined based on the particular needs of the application.

5.3 Implementation

There are three key issues in the implementation of batched futures:

1. how to represent batched futures in the veneer.
2. how to maintain the mapping between futures and actual objects in the database.
3. how to limit the size of the mapping

Futures have to be represented in such that a way that they are interchangeable with handles in stub objects. A mapping has to be maintained so that the database can associate a future used as an argument in a call with the corresponding result, and the size of this mapping has to be limited, since each call returns a new future and the mapping could potentially grow without bound. We consider each of these issues in turn in the following subsections.

5.3.1 Representing Futures

Futures are simply a special kind of handle, tagged to distinguish them from normal handles. In our implementation, we tag futures using the sign bit: futures are negative, handles positive. This convention is followed in our examples and diagrams.

Futures are chosen by the veneer, in a manner to be explained in the next section. Like handles, futures are not manipulated directly by the client but are encapsulated in a stub object, to which the client program is given pointers. Because of this level of indirection, there is only one instance of each future; on assignment, clients copy pointers, not the stub objects themselves, just as with handles.

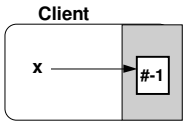

Figure 5-2: Representing Futures

5.3.2 Mapping Futures to Objects

Two parallel tables maintain the mapping between futures and objects, analogous to the H and VH tables that map handles to objects and stub object. The veneer future table VF maps futures to stub objects, and the database future table F maps futures to actual

objects. Slots in VF and F are initialized as a result of client calls, as we shall describe in the following subsections.

5.3.2.1 Maintaining VF

Slots in VF are initialized when the a stub function returns a future to the client. The stub functions themselves are unchanged when futures are used, but the function that reads the result object from the database is modified to choose a future instead and inform the database of its choice.

```
   stub_obj* get_object ()  // version using futures
   {
1     future_index = future_index + 1;
2     batch message to Thor: "assign call result to future future_index"
3     stub_obj* o = new stub_obj(-future_index);
4     VF[future_index] = o;
5     return o;
   }
```

Figure 5-3: Returning Futures

The veneer performs the following steps to create a new future, as shown in Figure 5-3:

1. increment a global counter to determine the index *i* of the future.
2. constructs a message to tell the database that the result of the last call should be mapped to future *i*. (The database will receive the message as part of the next batch of calls.)
3. allocate a stub object *o* that contains future *i*
4. store a reference to *o* in VF
5. return *o* to the client program.

Because the get_object function and the containing stub function are so simple (see Figure 5-3), they are small enough to be inlined into the client program, avoiding the expense of a function call.

5.3.2.2 Maintaining F

The database initializes slots in F when it processes a batch of client calls. Each batched call specifies the future index *i* that should be mapped to the result, as described in the previous section. After the database processes the call, it stores the resulting object in F[*i*]. When future *i* is used as an argument to a call, the server looks in F[*i*] to find the corresponding object. Because the server processes the calls in the same order they were made, the arguments of calls are determined by the time the server processes them.

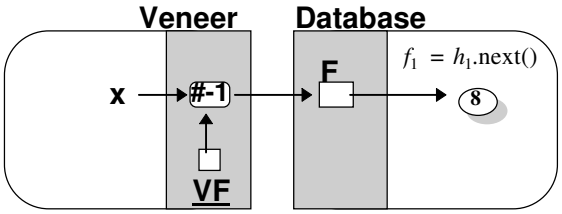

Figure 5-4: Mapping Futures to Objects

The following is a more detailed description of the steps performed by the database when it processes a call message from the client:

- Read the method name and arguments
- Look up each of the object arguments in H or F, depending on whether the index is positive or negative.
- Typecheck and perform the call
- If the result is a basic type or futures are not enabled, send the result back to the client, otherwise store a reference to the call result in F[*i*], where *i* is the index specified by the client.

Notice that the only overhead that has been added to the server stub is a couple of inexpensive conditionals. This overhead is minuscule and is dwarfed by the amount of time saved in avoiding a domain crossing and the amount of time spent performing the actual operation.

5.3.2.3 Example

Suppose *VF* is empty when the nth function shown in Figure 5-1 runs. After the for loop runs, futures 1,...,*n* will be assigned to results of the calls. Call *i* +1 will refer to the result of call *i* by using future *i*, and the slots of *VF* will point to the stub objects holding these futures.

When the calls run at the server, the slots in F will be assigned pointers to the objects corresponding to the results of the calls, and the object corresponding to the result of call *i* will be computed before it is used in call *i* +1. The operations inside the database never need to be concerned about the case where one of their arguments is an unevaluated future. The actual Thor operations called by the dispatcher require *no* modifications and run at their full speed.

5.3.3 Limiting the Size of the Future Mapping

Because each deferred call returns a new future and a future could be used as argument to any later call, tables *F* and *VF* can grow arbitrarily large.

Our implementation solves this problem by periodically replacing futures with the corresponding handles. After the number of futures passes a limit, the veneer flushes the pending invocations and piggybacks a request to the database to send the object handle equivalents for all futures currently in use. (The futures currently in use are the ones that have not yet been freed by the client.) The veneer uses the pointers in *VF* to update the stub objects. Because client copies pointer to the stub objects, and not the stub object themselves, we do not have to worry about updating and tracking multiple copies of each stub object.

The veneer and the database can then safely reclaim all slots in *F* and *VF*. Thus futures require only a constant amount of additional storage relative to normal calls.

Logically speaking, there is no reason why the database could not send back the corresponding handles after every batch of calls. However, there are performance advantages to sending handles in larger batches because it is cheaper to do one big read than a lot of small ones. Also, the longer the database waits, the greater the likelihood

that the client has freed futures in the batch, allowing the database to avoid sending back the corresponding handles, as discussed below.

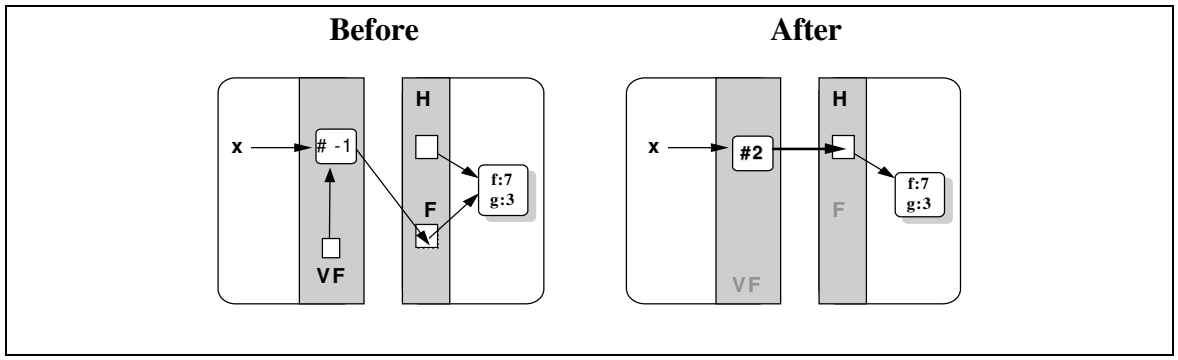

Figure 5-5: Remapping Futures

5.3.4 Stub Object Storage Management

Managing storage associated with stub objects containing futures is not a problem if the client language is garbage collected. After the veneer updates the future in a stub object with corresponding handle, it clears out the slot in VF. The stub object will not be discarded while still referenced from VF, but will be subject to garbage collection as soon as it is updated with the correct handle and the slot in VF is cleared.

As mentioned earlier, other languages require the programmer to use explicit deallocation. In such languages, the free_object operation should clear out the entry in *VF* as well as reclaiming the object storage. When the veneer attempts to replace futures with their actual values, it can simply skip the entries in *VF* that have been cleared, since otherwise it might overwrite memory that had been reallocated for other purposes. As an optimization, the veneer can tell the frontend at future remapping time exactly which futures have non-empty entries of VF, saving the expense of sending back handles for futures that have been freed.

There is one other concern regarding stub objects containing futures. When stub objects contain handles, the veneer almost always ensures that there is only one stub object for each database object. (It is possible that if an object migrates from one server to another, it will be assigned two or more different handles by the frontend. However this should be

a rare occurrence.) The same property *does not* hold for stub objects containing futures. This is a necessary consequence of the fact that the veneer does not know what the actual handle will be at the time the stub object is created. Therefore multiple stub objects can refer to the same database object. Such redundant stub objects are a problem only if the client has a very large number of *active* references that correspond to a small number of actual objects. (Inactive references are not a problem because in that case the stub objects will be freed.) Presumably, this will be not be a common occurrence. In general, we expect that most futures are only temporaries (that is, intermediate navigation pointers) and users won't want to hold onto them.

If redundant stub objects are a problem, the veneer can support an operation that takes a reference to a stub object, freeing it if it is redundant and returning a pointer to the corresponding 'canonical' stub object for that object's handle. (The operation requires, of course, that the client has no other references to the redundant stub object.) Alternatively, a customized garbage collector can do the same thing safely and automatically by redirecting links from the redundant object to the canonical one so that the former can be freed.

5.3.5 Shared Memory Optimizations

Batched futures can be implemented using Unix pipes. In this case the veneer just writes each message to the stream but does not flush the stream until it needs a result. However, it is faster to use shared memory if it is available. Each deferred call is recorded in a shared memory buffer, and data written by one side is always immediately visible to the other side without any need for flushes. (To obtain optimal performance, it is important that when one process blocks waiting for a result, the other is woken up quickly. Our implementation uses shared-memory semaphores to obtain this effect.) With this implementation, the database can begin working on deferred calls whenever it has time, even if the veneer is not yet waiting for the result of the last call in a batch. For example, on a multiprocessor this approach allows the database to process calls in parallel with the veneer.

Another shared memory optimization permits us to get rid of the *F* and *VF* tables and future remapping entirely. The tables exist only to allow futures used in later calls to be mapped to the results of earlier calls. The same effect can be achieved by allocating stub objects in shared memory. Rather than passing a handle or future, the veneer passes a pointer to the stub object, which the database dereferences. When the result of a batched call becomes available, the database stores the handle in the stub object allocated for the result, effectively performing the future remapping step for that object immediately.

The use of shared memory might raise the concern that the security of the system has been somehow compromised. Note, however, that the worst a client can do is overwrite a stub object in shared memory, which the client could do just as easily when the stub object was in its own private address space. The database is not exposing any more information than it was before, and its runtime typechecking will continue to prevent the client from making illegal calls.

5.4 Other Benefits of Batching Calls

Batching calls can provide other benefits in addition to amortizing domain-crossing overheads. Because the server can "see the future" by looking ahead in the current batch of calls, it can often improve performance based on that foreknowledge. Here are a few examples of how this might work:

- Thor could look forward in the call stream to see objects that will be used, and prefetch any objects that are not already in the cache. For example, in a breadth-first traversal, the client will make a set of calls that fetch each child of a known node. By looking ahead, the frontend can know that is definitely worthwhile to prefetch all of the children at once, rather than fetching each child individually as demanded by each operation.
- An interface to a filesystem could reorder read requests in a batch of calls to optimize disk head motion, and so forth.
- A three-dimensional rendering system could avoid performing an expensive rendering if it determined that a later call would obscure it.

Of course this ability to know future calls in advance has limitations. For example, if an argument to a call is a future, the system will not what it denotes until at least some of the preceding calls have been evaluated.

5.5 Exceptions

If exceptions are returned immediately, then no batching can take place. However, in the model we have proposed, in which the client makes explicit calls to check for exceptions in the exception history, there is a well-defined point, which need not be the point of the call, at which exception values are obtained. For purposes of batching, the veneer treats a request for an exception value in exactly the same way it would treat an operation that returns a basic value— batching stops and the current batch of calls is sent over to the database to obtain the exception value. To avoid unnecessary round trips to the database, the database sends over all of the non-null exception values that haven't been sent yet to the client, not just the particular one requested by the client.

Thus batching is possible assuming the client can defer asking for exception values until they are actually needed. Because exceptions propagate, as we mentioned, the client can often avoid checking for exceptions until the final call in a string of interdependent calls— the last call will succeed if and only all of the preceding calls succeeded. Thus checking for exceptions and achieving high batching factors need not be mutually exclusive aims.

Chapter 6.

Experimental Results

In this section we characterize the gains that can be expected from using futures, and present experiments showing the benefit obtained on various workloads. We begin with a simple mathematical model of the system's performance. We then exhibit systems that show the predicted performance across a range of batching factors and domain crossing costs. Finally, we give results from an OODB benchmark running on Thor that show that batched futures yield useful performance increases even in unfavorable conditions where only low batching factors can be achieved.

6.1 Performance model

The average cost of a call can be modeled by the formula

$$t_c + \frac{t_d}{B}$$

where t_c is the cost of running a call, t_d is the cost of the a pair of domain crossings, and B is the "batching factor": the total number of calls divided by the number of pairs of domain crossings. When there are no futures, $B = 1$ (since each call requires a pair of domain crossings); as we defer calls, B increases and the average cost of a call goes down. Note that the model assumes that t_d is independent of B, which is true in actual systems. In other words, the amount of time it takes to switch between domains is independent of the amount of data (that is, the number of batched calls) being transferred between the two domains.

The model predicts that, as B increases, the average cost of a call will asymptotically approach t_c, dropping rapidly at first and then with increasing slowness as t_d/B goes to zero and t_c begins to dominate the total cost of the call.

The key points to realize are that t_c provides a lower bound on the average cost per call, and that the larger the ratio of t_d to t_c, the more a system has to gain from batched futures, but also the higher the batching factors it must achieve before the domain crossing overhead t_d/B becomes negligible. For example, if the ratio of t_d to t_c is r, then a batching factor of r will yield performance that is within a factor of two of the optimal value.

6.2 Measured Performance: Best Case

The best case for batched futures is a client program in which all of the cross-domain calls can be batched. To experiment with a program that attains this best case, we considered the *nth* function described earlier. Each operation in the *nth* function returns a handle, so all of the calls can be batched, leading to an arbitrarily large batching factor B.

To show how the speedup provided by batched futures varies across a range of values for t_c and t_d, and also to demonstrate a range of systems to which batched futures are applicable, we considered three systems:

1. The first system is a simple client-server system with a very low t_c value and a comparatively high value for t_d. The client and server run in different process on the same machine and communicate using a shared memory buffer. The server, written in C++, implemented just the essential elements necessary to run the experiment: a linked list, a very simple dispatcher, a handle table, and a future table. It did not implement typechecking, garbage collection, concurrency control, persistence, or any of other features of Thor as a database. The stub functions and the client program were essentially the same as those in the case of Thor, however.

2. The second system is Thor running in its typical configuration, in which the client and frontend are running on the same machine but in different processes. The t_c for this system is significantly higher than that of the first system, in part because the functionality provided by Thor is much greater and in part because the frontend dispatcher is currently highly unoptimized.

3. The third system is again Thor, but with the client and frontend running on different machines. (It is possible that we will run Thor in this fashion in the case of client

machines with memory capacities too small for an effective frontend cache, or without sufficiently safe protection domains.)

The observed performance of each of the systems are shown in Figure 6-2 through Figure 6-4. The experiments were compiled using DEC C++ and run on a lightly loaded Alpha AXP3000 running OSF/1.3. Each of the Thor experiments ran with a warm cache, so that the Frontend did not have to go over the network to fetch objects. The average time per call was calculated by dividing the actual running time of the program by the number of calls on the database.

As predicted by the model, the average time per call drops rapidly at first and then approaches t_c for that system with increasing slowness; the shape for each graph resembles the graph of 1/B scaled by t_d and shifted up by t_c.

In the first system, batched futures lead to a greater than tenfold increase in performance for sufficiently large B, in the second only a threefold, and in the third, again about a tenfold increase in performance. The varying benefits of batched futures are explained by the differences in t_c and t_d across the different systems, as summarized in the Figure 6-1, which gives approximate values for t_c and t_d in microseconds for each system. The values for t_d were estimated by observing the difference between average call time for B=1 and B=2, which works out to be $t_d/2$. The value for t_c was estimated by subtracting t_d from the average call time for B=1, which is $t_c + t_d$.

	t_c	t_d
1	10	126
2	75	130
3	100	905

Figure 6-1: Values of t_c and t_d for some systems

We verified our estimates of t_c and t_d for Thor verified by running the *nth* function on a version of Thor in which the client and the database were linked into the same address space, allowing us to measure t_c directly (because t_d is zero.) In that case, we found that t_c averaged around 94 microseconds, in the same range with our estimates. It's interesting to note that with batched futures the average cost per call dropped to a value below this figure, to around 80. This is probably because batched futures eliminate the need to read a result handle after every call. Instead, a whole batch of future-to-handle mappings is a read at the same time, which is more efficient. The other interesting result is the fact that t_c is about 33% higher when Thor and the client run on different machines than when they run on the same machine. This is explained by the fact that the Unix calls to read and write from sockets are more expensive than those using our custom shared memory communication mechanism.

As mentioned, the maximum speedup for batched futures with Thor in its normal configuration (client and frontend on the same machine) is around 3. The speedup is small because operation marshaling, typechecking, and dispatching in Thor have not been optimized and are currently rather expensive, leading to a high value for t_c even though the operations themselves are simple. When the operation dispatcher is optimized, we can expect to see speedups from batched futures closer to those seen in the simple IPC system.

Even Thor's relatively large value of t_c is dwarfed in comparison with the cost of a network communication, as seen in the third system. In that case, batched futures again lead up to a tenfold speedup. If batched futures were used in a system that had the t_c of our IPC system and the t_d of a system communicating over a network, we would expect to see up to a 90 fold performance improvement. Obtaining this speedup, however, would require hundreds of operations being combined in each batch.

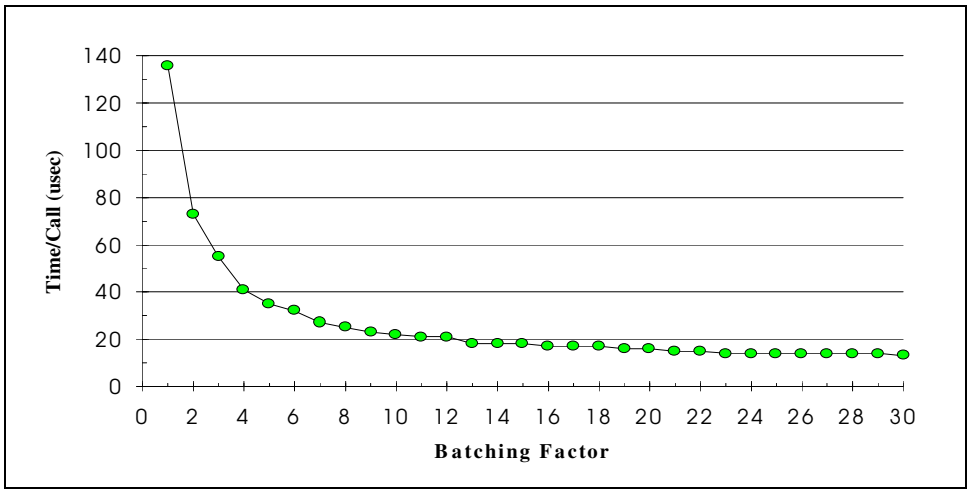

Figure 6-2: Best Case Performance— Simple IPC

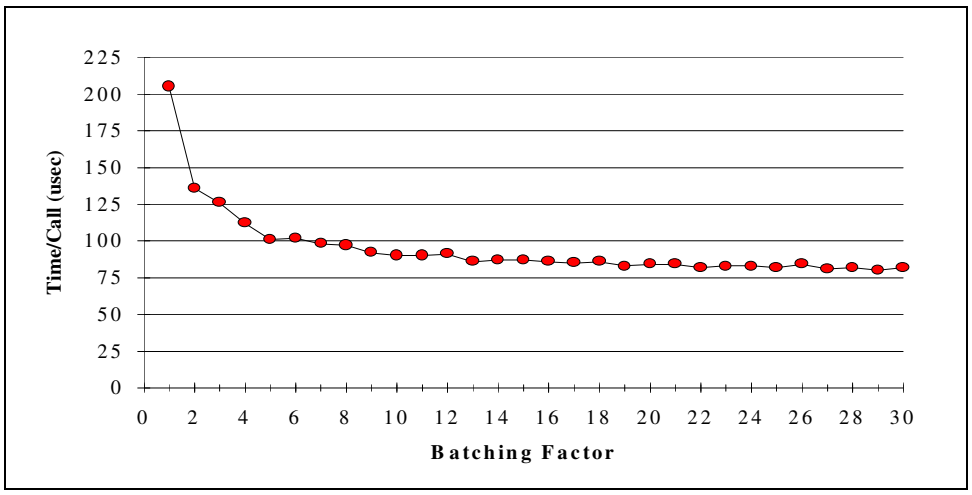

Figure 6-3: Best Case Performance— Thor, Local

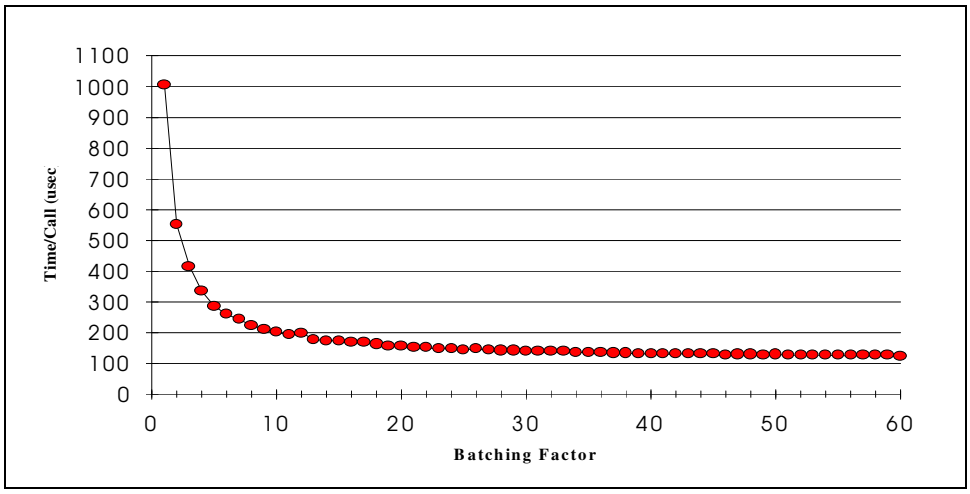

Figure 6-4: Best Case Performance— Thor, Remote

6.3 A Less Favorable Case

Not all applications are as favorable for batched futures as list descent. For example, in graph traversals, many values have to be known immediately, such as the number of nodes connected to the current node, meaning that many calls cannot be deferred and the batching factor is necessarily low. In this section, we give an example of such an application, and show that even under such unfavorable circumstances useful batching factors and performance increases still result.

As a representative application, we implemented various traversals from the OO7 suite of benchmarks for object-oriented databases [Carey 92]. The OO7 database consists of a set of interconnected parts, arranged in a hierarchy of complex assemblies, base assemblies, composite parts, and finally atomic parts. The operations on the parts and assemblies are quite simple: either returning a connected part, or returning a scalar attribute, such an integer id or a character documentation string. We implemented the parts and their primitive operations as types in the Thor database, and the traversals in the Thor C++ veneer using the methods defined by the OO7 type interfaces.

In practice, one might safely increase performance by implementing parts of the traversal inside of the database using Theta, the programming language for the Thor database. We wished, however, to assess the worst-case scenario, in which the client performs a large number of very fine-grained interactions with the database, many of which need to be performed immediately.

We ran OO7 traversal 2b, the traversal that demands the greatest number of fine-grained database calls. The traversal visits every part in the database, swapping the x and y coordinates of each atomic part. It makes a total of 207,399 calls on the database and has an actual running time of 46.19 seconds, for an average of 222 microseconds a call.

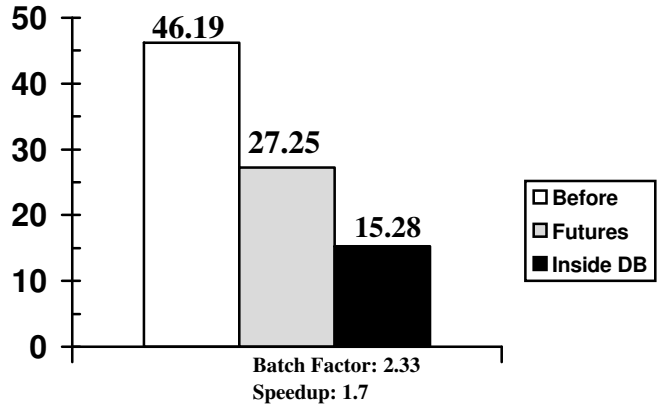

Figure 6-5: OO7 Traversal Performance

The batching factor was low, around 2.33, but not so low that batched futures did not yield useful performance increases. Even in a program such as the OO7 benchmark that needs to know many values immediately, the calls are about evenly balanced between operations that return objects and can be deferred, and calls that return values and need to be performed immediately.

As seen in Figure 6-5, batched futures increased the performance by a factor of 1.7, decreasing the running time from around 46 seconds to 27 seconds. How good is this improvement? To provide a basis for comparison, we measured the time for the traversal when the client program was linked into the database, so that the domain crossing overhead was zero but all of the other costs were the same. In that case, the time to perform the traversal was 15 seconds, so batched futures brought us to within a factor of two of the optimal time. The total time when running inside Thor is about a third of the normal value, which matches our measurements in the list descent case that show t_c is about a third of $t_c + t_d$, the total overhead of running an operation without futures.

Chapter 7.

Extensions

In this chapter, we introduce several extensions to the batched futures model that increase the amount of batching that is possible.

7.1 Futures for Basic Values

One obvious extension to batched futures is the ability to use futures for operations that would ordinarily return basic values. Sometimes the client does not need to know basic value results immediately, if at all. For example, the client can swap the values of two slots in an array of integers without knowing what those values are, or sum a set of values without knowing the values of all of the intermediate results, as long as the database keeps track of the intermediate values and allows the client to refer to them using futures.

The original, parallel version of futures allowed futures to be returned in place of basic values, while remaining completely transparent to the client programmer. However, this transparency came at a cost: the system had to add tags to every value to indicate whether it was a future or an actual value. Every time an operation depended on the actual value of an object, it had to check the tag to see if the object was a future and block if so until the actual value was available. These two requirements are incompatible with the demands of our veneer. Many client languages allow direct access to all of the bits of a value and will not tolerate tagging; furthermore inserting the necessary tag checks would require modifications to the client language compiler.

We therefore expose the distinction between futures for basic values and actual basic values to client programmers, allowing them to convert between the two forms on demand. (Our scheme is related, but not identical to the promises mechanism used with Mercury call streams; the small but significant difference is that futures for values can be passed as arguments to Thor operations without blocking.) Like promises, futures for

values are distinguished from normal values by making them a distinct type. For each basic value type T, the veneer defines a corresponding stub type Thor-T; we shall refer to such types collectively as ThorValues. (For example, the stub version of a client integer is a *ThorInt*. Similarly, the veneer defines ThorChars, ThorFloats, *etc*.) Conceptually, a ThorValue can be thought of as a pointer to a value that lives inside Thor. Each ThorValue type supports a 'dereferencing' operation that returns the corresponding client value; this is analogous to the *claim* operation for promises. Unlike promises, ThorValues also support a creation operation that encodes a client value as the corresponding ThorValue.

To allow futures for values to be passed as arguments without claiming them, the veneer type interfaces include "futurized" versions of each stub function that take and return ThorValues rather than basic values. For convenience, we also retain the standard versions of the stub functions, so that the client does not have to convert all client values into ThorValues before passing them as an argument to a Thor operation. If the client wishes to batch a call containing a mix of client and Thor values, it will need to convert the client values to Thor values. Notice that, as with handles, the veneer can batch up a number of interrelated calls without communicating with the database. Unlike with handles, the client has a way of obtain the actual representation of the return result.

A ThorValue is represented in the veneer as a union containing either an actual basic value, or a future for a call that will return a basic value, with a tag to indicate which is the case. If the client attempts to 'dereference' a ThorValue containing a future, the veneer sends the current batch of calls to the database.

To process a set of batched calls that use ThorValues, the database keeps a mapping between each future for a value and the corresponding basic value result, and sends back the actual results to the veneer in a batch. The veneer then overwrites the futures in the ThorValue stub objects with their actual values. Thereafter, the value for each of the ThorValues used in that batch of calls is available immediately without consulting the database.

As an optimization, the veneer can store the value of the future in some 'raw' form when it first gets it and defer the decoding until the client actually asks for the value. This allows the veneer to save the expense of decoding results that are only used as intermediate values by the client. For example, suppose the database and the client language use different formats for representing floating point numbers, and that it is expensive to convert between the two formats. If the client does not always need the values of the floating point numbers, the veneer could increase performance by performing the conversions lazily. (It might be advantageous to keep a copy of the original representation even after the number was converted, to avoid re-encoding the number if it is passed as an argument to another database operation.)

When futures are used for basic values, they are no longer completely transparent. The client must insert calls to obtain the client values corresponding to ThorValues when desired, and possibly also to wrap client values as database values. However, client programmers are always free to write their programs as usual with the normal version of calls, then convert them to use futures later as an optimization.

One concern is the proliferation of stub functions: adding futures for basic values has approximately doubled the number of stub functions. (The number is not exactly double because stub functions whose arguments and return values consist entirely of handles do not need a futurized version.) We can avoid substantially increasing the code size by inlining the stub functions, which is desirable in any event because they are small and fairly lightweight, especially when futures are being used. The other concern is that client programmers will be confused by a cluttered interface containing multiple versions of each stub function. We can minimize this problem by choosing sensible conventions for naming and ordering stub functions so that the futurized versions do not interrupt the flow of the interface. (The conventions we suggest are to use ALL-CAPS for the name of the futurized version of a stub function and to place them after all of the normal stub functions, where they can be easily ignored.)

We have not yet implemented futures for basic values. However, we have done some calculations of how they would increase the batch size for OO7 traversals and what effect

that might have on performance. Traversal 2b, the example we considered in this thesis, would benefit from the use of futures for basic values, were they available, because all of the calls to swap the integer x and y attributes can be deferred; the client has no need to know the actual x and y values.

Using promises, the average batch size increase from 2.33 to 3.47, and the predicted performance using the mathematical model increases from 1.7 times the standard performance to over 2 times faster.

7.1.1 Example

The following shows the normal and futurized versions of the fetch and store stub functions for an array of integers.

```
class IntArray {
    ...
    int         fetch(int slot) ;
    void        store(int  slot, int val);

    ThorInt*    FETCH(ThorInt* slot) ;
    void        STORE(ThorInt*  slot, ThorInt* val);
};
```

The interface first defines the normal versions of the calls, then the futurized versions, which take and return ThorValues wherever the original versions would take or return client values.

The client could use these functions to define a function that swaps two elements of an array with no additional domain crossings:

```
void swap(int_array *a, ThorInt* i, ThorInt* j) {
    /* Swap a[i] and a[j] using futures */
    ThorInt* a_i  = a→FETCH(i);
    a->STORE(i, a→FETCH(j));
    a->STORE(j, a_i);
}
```

7.2 Batched Control Structures

When a control structure in the client program depends upon the result of a database call, that call cannot be deferred, which limits the degree of batching that can be attained. For example, if an if statement depends upon the result of a call that returns a boolean, that call must be performed immediately, so that the correct branch in the client program can be chosen. What we would like to do is send the control structures along with the batched calls, and have them evaluated inside the database. If we could batch entire loops and conditionals, very high batching factors could be achieved.

Ultimately, we would like to be able to move arbitrary, *safe* pieces of the client program into the database protection domain. To do so in general is difficult and goes beyond the scope of this thesis; the veneer would need a detailed knowledge of the structure of the client program and would essentially become a compiler or interpreter for the client language. However, for certain restricted but useful cases, it is quite simple to allow batches to include control structures.

Consider the following example, which increases the salary of all managers by $1000 and all other employees by $500.

```
Array<th_employee*> a = employees→elements_array();
for (int i=0; i < a.size(); i++) {
    th_employee* e = a[i];
    if (e→is_manager())
        e→increase_salary(1000);
    else
        e→increase_salary(500);
}
```

Figure 7-1: Client Control Structures

The code uses the elements_array function to obtain the members of the employees collection, and increases the salary of each employee by either 1000 or 500 depending on the results of the is_manager() call.

Notice that the code, except for the control structures, consists of *deferrable* calls on the database. These conditions make it possible to batch the control structures along with the calls. The basic idea is that instead of using client language control structures, the

application uses veneer calls that represent the control structure. These calls are deferred; they include the names and arguments to the control structure, allowing the database to evaluate the control structure itself. In C and C++, the new control structures are implemented as preprocessor macros, so that they have the syntactic form of control structures even though they are actually calls.

7.2.1 The Meaning of Batched Control Structures

```
employee *empl;
FOREACH(empl, employees→elements())  {
    IF (empl→IS_MANAGER()) {
        empl→increase_salary(1000);
    }
    ELSE {
        empl→increase_salary(500);
    } ENDIF
ENDFOR
```

Figure 7-2: Batched Iterator and Conditional Control Structures

In Figure 7-2, we show how the Figure 7-1 might be rewritten using batched control structures. It uses two different batched control structures: FOREACH and IF. The FOREACH control structure is used to call a Thor iterator and takes two arguments: a future for the iterator that will control the loop, and a loop variable that will refer via a future to the value yielded by the iterator each iteration. The ability to call iterators directly, rather than obtaining their results as an array, is a new feature made possible by the fact that the entire FOREACH control structure is batched and evaluated inside the database. The IF statement takes one argument: a future for a boolean that will control the branching in the database. The remaining structures, *i.e.* ELSE, ENDIF and ENDLOOP, take no arguments and simply batch a call to indicate where the first matching unclosed control block ends. They are necessary because the database has no other way of knowing when a block ends, since it has no access to the actual client code.

The stub functions for the batched control structures in the veneer work essentially the same as any other stub function. The FOREACH stub function batches a message to the database with the future for the iterator and a new future to be mapped to the yielded

results. Similarly, the if stub function batches an "if" call to the database with the future for the conditional. The elements() stub function, which returns a future to an iterator, and all of the other deferred calls, work exactly as before. In no case, however, is the control flow of the client program altered. Note that all the calls, including the conditional test, are deferred; we are assuming that the veneer is using futures for basic values as well as objects, as described in Section 7.1.

The resulting set of batched calls sent to the database is depicted in Figure 7-3. We use the conventions here that we have used in earlier diagrams: h represents the handle for employees, and f_i represents the ith future allocated in the course of the example.

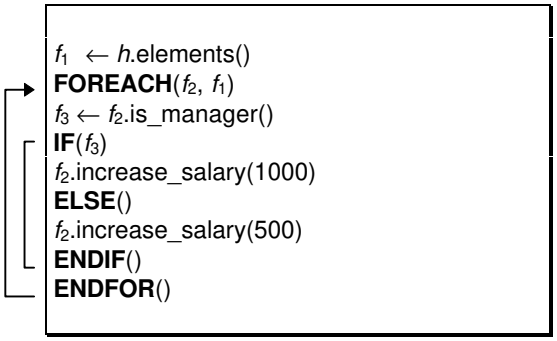

Figure 7-3: Batched Calls and Control Structures

7.2.2 Restrictions on Batched Control Structures

Several constraints on the use of batched control structures follow from the semantics of the deferred code. These constraints stem from two essential facts:

- every statement in a block is executed exactly once on the client side, whereas in the database the same block may be executed zero or many times.
- the database knows only the future mapped to a call result, not the *name* of the client variable to which the future is assigned.

From the first fact it follows that there must be no side-effects to *non-Thor* client variables within a deferred control structure. Because the side-effects are not evaluated in Thor, it is not possible for the database to provide the desired semantics of conditional or repeated evaluation for client side-effects. If side-effects are desired, the client can achieve them by

manipulating Thor objects and values. Rather than incrementing a client integer, for example, the client can increment a ThorInt and get the actual value after the loop has completed.

More importantly, from the first and second facts follows an "assign-once" restriction on client identifiers. The identifiers used to refer to futures in batched control structures just allow the results of particular calls to be used in later calls; they are not variables. For example, consider the code on the left hand side of Figure 7-4. The meaning of this code is that, after the IF, v refers to the future of the call of the call in the ELSE branch, regardless of which branch is taken when the code runs in the database. In general, identifiers obey the following scoping rule: the meaning of a use of an identifier is always the future assigned to that identifier in the invocation statement most immediately preceding that use, *disregarding control structure*. If the branch of the computation containing that invocation statement is not taken, the identifier may not mean anything at all— its slot in the future table can be null.

Therefore all of the identifiers in a batched control structure need to be "assign once", that is, used as the target of just one assignment. To ensure this, client programs should follow the convention of using a fresh variable name for each assignment of the result of a Thor call to a variable. This may make it necessary to duplicate some code in the case of conditionals, as seen in the example on the right hand side of Figure 7-4. Because of the copying of code into both branches of the statement, these transformations could potentially lead to exponential blowup in the size of the client code, certainly an inconvenience even though technically no expressive power has been lost.

IF (...)
  v = f();
ELSE
  v = g();
ENDIF
u = v→foo()
u→bar()
	IF (...)
	  v_{then} = f();
	  u_1 = v_{then}→foo()
	  u_1→bar()
	ELSE
	  v_{else} = g();
	  u_2 = v_{else}→foo()
	  u_2→bar()
	ENDIF

Original Code	***"Assign-Once" Code***

Figure 7-4: The Assign-Once Restriction

As a result of the assign-once restriction, variable assignments cannot be carried over from one iteration to the next, since to do so would require two assignments: one before the first use, and a second to change the variables value after that use.

A final restriction on batched control structures is that there is no mechanism provided for batched recursion. This is mostly an inconvenience, because the same effect can be achieved using an explicit stack.

7.2.2.1 Example

It might seems with all of these restrictions that it would awkward at best to write useful batched control structures. To show that this is not the case, we present batched control structures that perform a depth first traversal of a graph in Figure 7-5. The program uses a queue stored in Thor to simulate the recursion; otherwise no changes from natural coding style are required. (We introduce a batched WHILE statement to make the code more natural; WHILE is implemented along the same lines as any other batched control structure.) Since the OO7 benchmark consists of various kinds of traversals, it would probably be feasible to implement OO7 entirely in terms of batched control structures, leading to very high batching factors.

```
void DFS(th_node* n) {
    th_queue *q = th_queue_type→new();

    q→push(n);
    WHILE (q→empty()→IS_FALSE()) {
        th_node* current = q→dequeue();
        //... perform some action on current

        th_node* child;
        FOR_EACH(child, current→children()) {
            q→enqueue(child);
        } END_FOR
    } END_WHILE
}
```

Figure 7-5: Batched Depth First Traversal

7.2.3 Evaluating Batched Control Structures

We now consider the trickier part of batched control structures: evaluating them inside the database. At a high level, the database first typechecks the deferred code and then runs it.

To do so, the database first processes the code into a tree form that expresses its structure. The task of processing the deferred calls is very much like writing a simple compiler or translator. The input language is a simple linear stream describing the deferred calls; the output language is a tree describing the "program" that accomplishes those calls. The form of this tree is like an ordinary parse tree; that is, it has a different kind of node for each statement type and a node has subnodes as needed for that type. For example, a FOREACH node would have three subnodes: iter, which indicates the iterator that controls the loop, loop_var, which gives the future table index for the slot in which the loop variable is stored, and body, which points to a body statement node containing the code of the loop body.

7.2.3.1 Typechecking

As with standard batched futures, typechecking uses slots in F to store intermediate results. In this case we store both the value of the future (i.e., the object it will refer to when the program runs) and its type. For example, in the example above, slot f_2 holds the loop variable and slot f_3 holds the result of evaluating the call on method is_manager(). Typechecking the code involves a linear scan of the batch of deferred calls. When the typechecker encounters the assignment to a given future table slot, it is always the case that of the futures in the operation being typechecked have had their types determined already. The database uses the signature of the called operation to determine what type will be returned and it stores this type in the future table slot for that future. For example, the elements signature allows Thor to infer that f_1 is an iterator that returns employees, and hence that f_2 will hold pointers to employee objects. Typechecking in this fashion is correct because each future represents the result of exactly one call. If futures could represent the result of more than one call, things would be quite a bit more complicated. For example, a future might enter a loop with the correct type for its initial use, then be modified after that use to contain an object of an invalid type, so that in the next iteration type-safety is violated. However, multiple assignments to the same future are not possible under the current design. Regardless of the *identifier* assigned to the result of a call, different calls *always* use different futures.

7.2.3.2 Evaluation

Evaluating the program tree is straightforward; effectively, Thor acts as an interpreter of the tree. For example, at a for node Thor runs the following code:

```
eval_for (s: for_stmt) returns ()
   for a:any in s.iter() do
         F[s.loop_var()] := a
         eval_body(s.body())
    end
end eval_for
```

The code runs the requested iterator, assigning each value that it yields to the future table slot specified in the for statement node. It then evaluates the body. Similarly, evaluation functions for other control structures evaluate the appropriate blocks based on the provided arguments.

7.2.4 Additional Benefits of Batching Control Structures

One obvious benefit of batched control structures is the greatly increased batching factors that they permit. For example, it seems that an entire OO7 traversal, or at least substantial pieces of it, might be written as a single batched control structure. The entire traversal would require no extra communications with the database.

In addition to increasing the batching factor B, batched control structures also reduce the call cost t_c. They allow the set of calls in a loop sent to the database to be sent over only once rather than once per iteration, increasing performance by reducing the amount of data the database has to read. They also amortize the cost of typechecking and other expensive components of t_c.

Finally, the number of future table entries needed for the loop is greatly reduced. Each iteration, the calls in the body are reevaluated with the new value for the iterator future, and the old futures for the calls are mapped to the new result. Thus, regardless of how many times the loop executes it uses the same number of futures.

7.2.5 Comments

Batched control structures probably represent the limit of what can be achieved without actually inspecting the client program. They already impose a perhaps excessive burden on the client programmer to follow the restrictions outlined above. However, the increased batching, decreased typechecking overhead, and other performance advantages they allow will justify their use by experienced client programmers in some cases. Based on our measurements, typechecking and domain crossing overheads take up over 90% of the time required for each call; batching entire loops will amortize these overheads to negligible amounts for reasonable sized loops, leading to an order of magnitude performance increase in realistic cases. Furthermore, in the future work section of Chapter 8, we discuss an approach to eliminating the greatest restriction on batched control structures, the "assign-once" restriction.

Chapter 8.

Conclusions and Future Work

This thesis has described the design and implementation of Thor veneers, a safe, language independent interface to an object-oriented database, and batched futures, a general mechanism that reduces the cost of client calls to servers that run in a separate protection domain. Futures allow calls to be deferred until a client really needs a result and then made all at once in a batch; later calls in the batch can refer to the results of earlier calls.

The work was done in the context of Thor, but can be used in other systems. Our implementation depends on knowing whether a call returns an opaque pointer or a value and can be used in any system where this information is available, for example, an operating system. Even without this information, an approach in which the client chooses whether or not to defer a call is always possible. The mechanism can be used when the client and server run in different processes on the same machine, and also when the client runs at a different machine than the server.

The thesis analyzed the performance gains that can be expected from our mechanism and presented performance results to show that futures yield useful improvements. As part of our performance studies, we analyzed the cost of making cross-domain calls and developed a formula that accounts for the cost. The formula has two components: one for the domain crossings, and one for performing the call. Obviously the relative costs of the components will vary for different systems and for different workloads, but we believe that the following statements are true in general.

First, the cost of domain crossings is likely to increase relative to the other costs because of current developments in computer architecture. For example, pipelining and superscaler processors depend upon being able to look ahead in the instruction stream. In general, looking ahead is difficult if not impossible to do if the following instructions are in

a different address space or protection domain, so domain crossings are likely to lead to increased processor stalls and cache misses. Second, although some workloads might be restricted to calls such that it takes a long time just to do the work of the operation, many workloads will not have this property. Therefore we conclude that our results are of interest generally.

Although significant benefit can be obtained by using futures even when batches are small, speedups are limited by the number of operations that can be deferred. Even with futures for basic values, many of the OO7 benchmarks do not allow deferring very many calls, because after a few calls the application needs to do something itself, *e.g.*, look at a value to determine what to do next.

To do better requires a way of getting longer chains of deferred calls. We presented a scheme that allowed the client to combine calls with simple control structures in a batch. The database interprets the structures and only carries out the calls on the indicated paths. However, our scheme had limitations, most notably restrictions on side-effects to client variables within the batched control structures and an inability to handle recursion. An interesting direction we intend to pursue is to eliminate the "assign-once" restriction by making assignment a deferred operation as well. By manipulating the mapping of futures to handles, the database can achieve the same effect as an assignment to a client variable, allowing it to perform only the assignments actually encountered in the evaluation of a batched control structure. This would make the semantics of batched control structures identical to those of client control structures, as long as no side-effects to non-Thor client variables were included.

A more flexible alternative is suggested by Stamos' work on *remote evaluation* [Stamos 86] , in which a client procedure is sent to be evaluated in the server address space. However, remote evaluation as proposed by Stamos supported only a single, safe language, avoiding the issues of safety and heterogeneity that are important in our work. For the mechanism to work safely, the remotely evaluated procedure must not be able to violate the security of the database. This implies that it must be written in a "safe" subset

of the client language. Probably not all languages have useful safe subsets but looking for them is an interesting research direction.

One possibility for ensuring that the veneers only attempt to remotely evaluate safe subsets of the client language is to require that they translate the subset into Theta, which is safe by design. For many languages, this is not significantly more difficult than parsing the program and verifying that it is safe. In addition, it solves the problem of needing an interpreter for every client language that might be remotely evaluated in the database, since the database can dynamically link Theta code into itself. If an entire application could be written in the safe language subset, it could run inside the database without any domain crossing or checking, thereby achieving the best performance.

Works Cited

[Adya 94] Atul Adya. *Transaction Management for Mobil Objects using Optimistic Concurrency Control*, Masters Thesis, MIT, January 1994.

[Barrera 93] J. Barrera. *Invocation Chaining: Manipulating Lightweight Objects across Heavyweight Boundaries*, Fourth Workshop on Workstation Operating Systems, October 14-15, 1993, pp. 191-193.

[Bershad 90] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. *Lightweight Remote Procedure Call*, ACM Transactions on Computer Systems, 8(1), February, 1990.

[Birrel 83] A.D. Birrel and B.J. Nelson. *Implementing Remote Procedure Calls* , Xerox CSL 83-7, October 1983.

[Birrel 94] A.D. Birrel et. al. *Network Objects*, Digital Equipment Corporation, SRC Research Report 115.

[Black 87] A. Black, E. Hutchinson, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE TSE*, pp. 65-76, January 1987.

[Butterworth 91] P. Butterworth, et. al. *The Gemstone Object Database Management System*, Communications of the ACM, Volume 34, Number 10, October 1991, pp. 65-77.

[Cardelli 88] Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138-164, 1988

[Carey 92] M. Carey, D. DeWitt, and J. Naughton. *The OO7 Benchmark*, Work in Progress, October 26, 1992.

[Day 94] M. Day, R. Gruber, B. Liskov, and A. Myers. *Abstraction Mechanisms in Theta*, in prep.

[Getty 90] J. Getty et al. *The X window system version 11*, Digital Equipment Corp., Cambridge Research Lab, December 1990.

[Gifford 86] D. Gifford. *Remote pipes and procedures for efficient distributed communication.*, MIT/LCS/TR-384, October 1986, p. 24.

[Goodenough 75] J. Goodenough. Exception Handling: issues and a proposed notation. *Communications of the ACM* 18(12): 683-696, 1975.

[Helfinstine 94] B. Helfinstine. Bachelors Thesis (in prep), MIT Department of Electrical Engineering and Computer Science, 1994.

[Halstead 85] R. Halstead. *Multilisp: A Language for Concurrent Symbolic Computation*, ACM Transactions on Programming Languages and

Systems, Volume 7 Number 4, October 1985.

[Hwang 94] D. Hwang. *Indexing for Fast Associative Access to Large Object Sets*, Ph. D. thesis, MIT Laboratory for Computer Science, 1994.

[Lamb 91] C. Lamb et. al. *The ObjectStore Database System*, Communications of the ACM, Volume 34, Number 10, October 1991, pp. 51-63.

[Liskov 81] B. Liskov et. al. *CLU Reference Manual*, New York: Springer-Verlag, 1981.

[Liskov 88a] B. Liskov. *Communication in the Mercury System*, Proceedings of the 21st Annual Hawaii Conference on System Sciences, IEEE, January 1988, pp. 178-187.

[Liskov 88b] B. Liskov and L. Shrira. *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*, Proc. ACM SIGPLAN '88, June 1988.

[Liskov 90] B. Liskov. *A highly available object repository for use in a heterogeneous distributed system*, Proceedings of the Fourth International Workshop on Persistent Object Systems Design, Implementation, and Use, pages 255-266, Martha's Vineyard, MA, September 1990.

[O Deux 91] O Deux et al. The O_2 System, Communications of the ACM, Volume 34, Number 10, October 1991, pp. 35-48.

[Schaffer 85] C. Schaffer, T. Cooper, C. Wilpolt. Trellis: Object-based environment language reference manual. Technical Report 372, Digital Equipment Corp./Eastern Research Lab., 1985.

[Stamos 86] J. Stamos. *Remote Evaluation* , Ph.D. Thesis, MIT Laboratory for Computer Science, Technical Report 354, January 1986.

[Stroustrup 87] B. Stroustrup. The Evolution of C++ 1985 to 1987. In *Proc. Usenix C++ Workshop*, pages 1-22. Usenix Association, November 1987.

[Wahbe 93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, *Efficient Software-Based Fault Isolation*, Computer Science Division, University of California, Berkeley.