Code Importing Techniques for Fast, Safe Client/Server Access

Joseph A. Bank

September 1996

© Massachusetts Institute of Technology 1996. All rights reserved.

This work was supported by a grant from Sun Microsystems.

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

Code Importing Techniques for Fast, Safe Client/Server Access

Joseph A. Bank

Abstract

In client/server systems, server integrity can be maintained by disallowing direct client access to server data and requiring communication through well-defined interfaces. Typically, this communication requires expensive cross-domain calls, or even more expensive network communication. The main goal of this thesis is to create a system with fast, safe client/server access by allowing the server to import client code. Importing code allows the client to access the server through direct procedure calls, which are much faster than the indirect access that is typically used to maintain integrity. Server integrity is maintained by using a type-safe client language and by verifying imported code.

To explore the benefit of allowing the server to import code for client/server communications, a Java interface to the Thor object-oriented database has been implemented, using code importing techniques for safe database access. Performance benchmarks indicate the use of code importing in the Java interface to Thor performs 50 times faster than the Java interface to Thor using standard communication techniques. These benchmarks also indicate that the Java interface to Thor outperforms by a factor of 3 a highly optimized C++ interface to Thor. However, since Java is interpreted, and thus an order of magnitude slower than most compiled languages, imported Java code is outperformed by code written in the language of the database itself. In the future, the use of just-in-time compilation should improve the performance of Java, making code-importing even more competitive.

Keywords: client/server, code importing, object-oriented database, Java, compiling, code verification, safe programming languages

Acknowledgments

I would like to thank my advisor, Barbara Liskov, for providing assistance and valuable suggestions. I would also like to thank Ron Bodkin, Andrew Myers, Quinton Zondervan, and Miguel Castro for reading early drafts and providing valuable feedback. Sylvia Chen also deserves thanks for putting up with me while I was working on my thesis.

My thanks also go out to Sun Microsystems for providing the source code for the Java runtime and more importantly for providing the funding for this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many distributed applications use a client/server structure to both divide the computation and to isolate the server from client errors. In these client/server systems, server integrity can be maintained by disallowing direct client access to server data. The client communicates with the server using a well-defined interface that provides limited access and error checking. The main drawback of these types of client/server systems is that the indirect client/server communication is expensive. A number of different approaches to this performance problem have been suggested. One approach is to trade-off safety for performance. By allowing the client to access the server data directly, increased performance is achieved by sacrificing server safety. Alternative approaches have aimed at reducing the communication cost between the client and server when safety is critical. The communication cost reduction is achieved by either reducing the cost associated with a single message or reducing the number of messages needed.

This thesis presents a client/server approach that gives high performance without sacrificing safety. By importing and running client code in the server, the client code can access the server through direct procedure calls. Server integrity is maintained by using a type-safe client language and by verifying imported code.

1.1 RPC

In typical client/server systems, the client and server run as separate processes and use interprocess communication (IPC) mechanisms to communicate. The most common type of IPC used in client/server systems is some form of Remote Procedure Call (RPC), which

provides the client with a procedural semantics for interacting with the server. Figure 1-1 illustrates the main components of RPC. Using RPC, the client calls a *stub routine*, which first marshals the arguments (i.e., translates them into a form understood by the server), and then uses a transport layer to send the requested call to the server. The server also uses a stub that unmarshals the arguments, performs the call, and returns the results to the client stub using the transport layer. The client stub then unmarshals and returns the result of the call to the client.

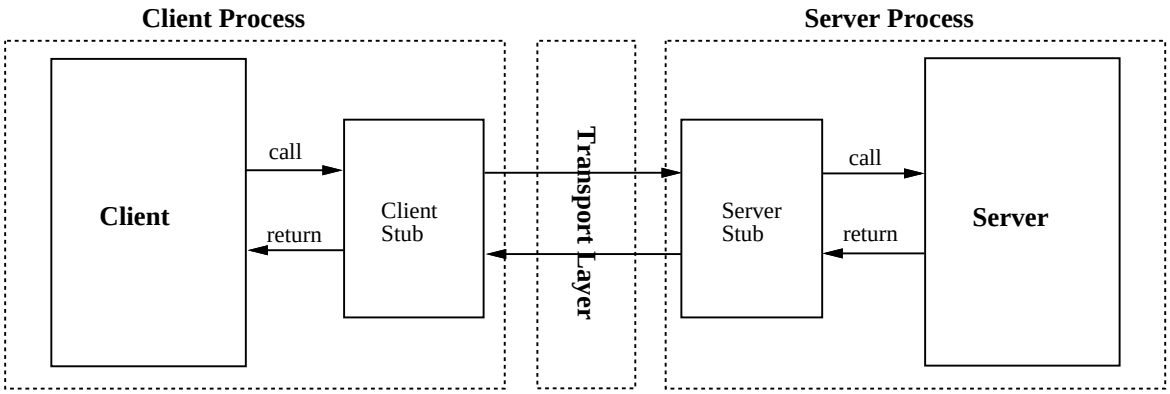

Figure 1-1: Remote Procedure Call

Using RPC has three important advantages. First, RPC provides the client with familiar semantics that are identical to those of a procedure call. Another advantage of using RPC is that it provides heterogeneity. By having the client and server stubs convert between representations and communicate using a well-defined interface, the client and server can use different languages and different data representations (e.g., big-endian and little-endian). A final crucial advantage of using RPC is that it allows the server to protect itself from a misbehaving client. Since the client and server run in different address spaces, the client cannot directly access server data. By limiting the client's interaction with the server to RPC and performing dynamic type-checking in the server stubs the server can protect itself from accidental (or intentional) errors in the client.

In an RPC system there are a number of fundamental costs beyond the cost of the actual procedure call. The main costs are marshaling, unmarshaling, type-checking and validating the arguments, as well as the cost of the communication between client and server. The performance of client/server system using RPC is poor when the client makes a large number of RPC calls that each do only a small amount of work. When the work

performed by each individual RPC call is small, the overhead of using RPC can easily dominate the computation time.

1.2 Code Importing

The main goal of this thesis is to create a client/server interface that keeps many of the benefits of RPC while improving performance. The client/server interface uses a form of function shipping called *code importing* to download client code to run directly inside the server. Figure 1-2 illustrates how code importing works. First, the client code is sent to the server. The server incorporates the client code into the server process. The client code then makes calls to stubs, which provide the interface between the imported client code and the server. These stubs combine the functionality of the client and server stubs of RPC. The stubs marshal the arguments, make a direct call to the server, unmarshal the result, and return it to the client code.

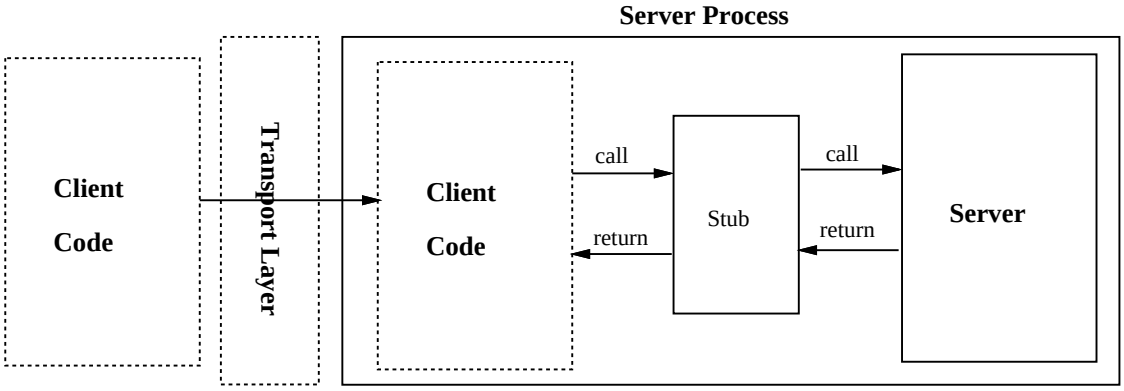

Figure 1-2: Code Importing

This code importing design has a number of advantages. The most important is that once the code is imported, there is no longer a performance penalty associated with calls to the server. Since the client code runs in the same process as the server, IPC techniques that require expensive cross-domain calls or even more expensive network communication are not used. Instead a simple procedure call can be used. Another advantage is that the use of stubs provides the same heterogeneity as RPC; the client code can use a different language and different data representation from the server.

Since the client code runs in the same address space as the server, the server's integrity is not guaranteed to be protected from a misbehaving client. To achieve the same safety

that can be provided by an RPC system a type-safe client language is used along with code verification when the code is first imported. These techniques guarantee that the client can only access the server through the stubs provided by the system. Additionally, since the client language is type-safe, the stubs do not have to perform dynamic type-checking.

Code importing presents some fundamentally different performance trade-offs from RPC. The main overhead in the code importing system are the initial downloading and verifying of the client code and the marshaling and unmarshaling of arguments in the stubs. Because downloading and verifying code is fairly expensive, the initial cost of using code importing is large compared to a single RPC call. But subsequent calls from the imported client code to the server are extremely inexpensive compared to an RPC call. Code importing shows improved performance over RPC when the initial overhead is amortized over a large number of calls to the server. Since code importing provides a more expressive interface than the simple procedure call interface of RPC, code importing makes it easy to package a large amount of work for the server into a single call. For example, an entire client application can be downloaded to run directly inside of the server.

To evaluate the benefits of using code importing, I implemented two interfaces between Thor[LDG+96], a new object-oriented database system, and clients written in the Java programming language[Sun95a]. The first Java interface to Thor uses a variant of RPC techniques for safety, communicating with Thor using TCP/IP communication. The second Java interface to Thor imports and runs the Java client code inside the Thor server using an embedded Java interpreter.

Performance analysis indicates that the use of code importing in the Java interface to Thor provides a performance improvement of a factor of 50 over the RPC based Java interface to Thor that uses TCP/IP for communication. These benchmarks also indicate that the Java interface to Thor outperforms by a factor of 3 a highly optimized RPC based C++ interface to Thor that uses a shared memory buffer for IPC. Yet, because Java is interpreted, and thus an order of magnitude slower than comparable native machine code, importing Java code is far outperformed by code written in the language of the database itself. In the future, the use of just-in-time compilation should allow Java performance, and thus the code-importing Java interface to Thor, to improve substantially.

1.3 Roadmap

The remainder of this thesis is divided into seven chapters. Chapter 2 gives some background on the Thor object-oriented database system and the Java language. Chapter 3 introduces the client API (Application Programmer Interface) provided by the Java interface to Thor. Chapter 4 describes the design and implementation of a Java interface to Thor that uses standard RPC client/server techniques. Chapter 5 describes the implementation of a Java interface to Thor that uses code importing. Chapter 6 gives performance results and analysis from running an object-oriented database benchmark with Thor using both standard RPC client/server techniques and code-importing. Chapter 7 presents some comparisons with related work. Finally, Chapter 8 provides some possible extensions to the work of this thesis and presents some conclusions that can be drawn from this work.

Chapter 2

Background

This chapter gives a brief background on the Thor object-oriented database system and the Java language. For a more complete background on Thor, see [LDG+96, LAC+96, LCD+94]. For a more complete background on Java, see [Sun95a, Sun95b, Yel95].

2.1 Thor

Thor[LDG+96] is an object-oriented database being developed at MIT that provides persistent and highly available storage for its objects. Thor objects are encapsulated, requiring clients to access database objects through calls of object methods and routines. Thor provides a good fit for experimenting with code importing instead of RPC, since the standard client interface to Thor is based on a slight variant of RPC.

2.1.1 Theta

Objects in Thor are implemented using the Theta language. Theta is a strongly-typed language that guarantees that objects are used only by calling their methods. In addition, all built-in Theta types do runtime checks to prevent errors, e.g., array methods do bounds checking. Theta is based on a heap with automatic storage management. More information on Theta can be found in [LCD+94].

2.1.2 Language-Independent Interface

The Thor system provides a language-independent interface based on RPC to allow clients to access the Thor server. In RPC terms, the language-independent interface provided by

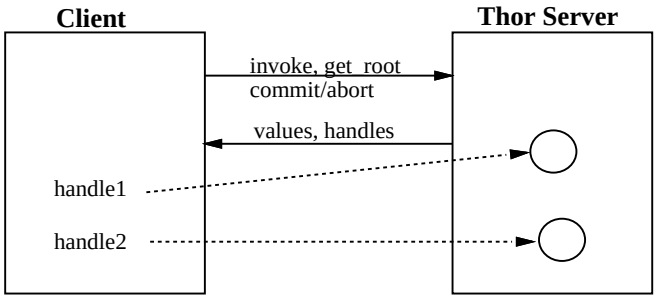

Figure 2-1: Thor Language-Independent Interface

Thor consists of the server-side stubs and the communication protocol for using those stubs from a client. To use the language-independent interface from a particular client language client-side stubs that use the protocol provided by the language independent interface need to be generated.

Safety

The protocol and server stubs of the language-independent interface do not make any assumptions about the client language. Certain client languages are unsafe (e.g., C and C++), and cannot be trusted with direct access to Thor objects. Thus, Thor objects and the code for object methods are stored and run inside the Thor server. To protect the integrity of data stored by Thor, the language-independent interface requires that clients run in a separate address space from the Thor server, communicating using the RPC techniques described in Chapter 1.

As Figure 2-1 indicates, clients using the language-independent interface do not store direct pointers to Thor objects. To refer to an object the client uses an opaque pointer called a *handle*, which Thor maps back to an object. The use of handles allows Thor to check the validity of every object referenced by the client. Thor clients can also use languages that are not type-safe (e.g., C or C++). To protect Thor from client type errors, the language-independent interface's server-side stubs perform runtime type checking. For every method call from a client, Thor checks that all handles refer to a valid objects, that a valid method is being called, and that the number and types of the arguments are correct. The use of type-checking and validation in the server-side stubs combined with running the client and server as separate processes ensure that the Thor server is protected from client errors.

Functionality

The protocol used by the language-independent interface provides the functionality needed for Thor clients to interact with Thor. Clients issue to Thor one of a number of commands, such as *get_root*, *invoke*, *abort*, and *commit* along with additional information needed by the given command. For example, for the client to call a method on a Thor object, the *invoke* command is issued, followed by the object's handle, a method handle indicating what method is to be invoked, and any additional arguments needed by the method (either handles or basic values such as integers). When Thor receives the client message, it examines the type of the command and dispatches to code that handles method invocations. At this point, arguments are type-checked and handles are mapped to actual pointers to Thor objects. Thor then performs the method call. The response from Thor indicates if the command was successful or caused an exception. If the client command requires a return value and did not cause an exception, a return value (a handle or basic value) is also sent.

The language-independent interface also includes a mechanism for garbage collecting client references to Thor objects. The Thor system is garbage collected; thus memory is reclaimed when the system determines that a particular object is no longer in use. The Thor garbage collector takes into account that the client may hold a reference to an otherwise unused object. Whenever Thor returns an object reference to the client, Thor assumes that the client references that object until it is told otherwise. To allow objects referenced by the client to be garbage collected, the client periodically sends Thor a list of handles that it no longer uses. Relying on the client to free objects is dangerous since the client may try to reuse a reference to a freed object. However, since Thor checks the validity of all handles, this kind of error will be caught by Thor and will not disrupt system integrity (although it would indicate an error in the client implementation).

Veneers

Thor is designed to allow clients to be written in many different languages. A *veneer* provides the functionality necessary to allow a new client language to be used. A veneer is a library of client-side stub routines that implements the client-side of the protocol provided by the language-independent interface. Aside from simply implementing the protocol, a veneer is responsible for mapping between the features of Thor and the features of a given language.

A veneer translates between basic types stored in Thor (e.g., integers) and related types in the client language. The veneer also provides a means of interacting with Thor objects. An automatic stub generator uses the Theta type definition to produce a corresponding stub type in the application language together with operations that correspond to methods of the Theta type. Currently, Thor veneers have been written for C, C++, Perl and Tcl. Chapter 4 discusses the implementation of a Java veneer based on the language independent interface.

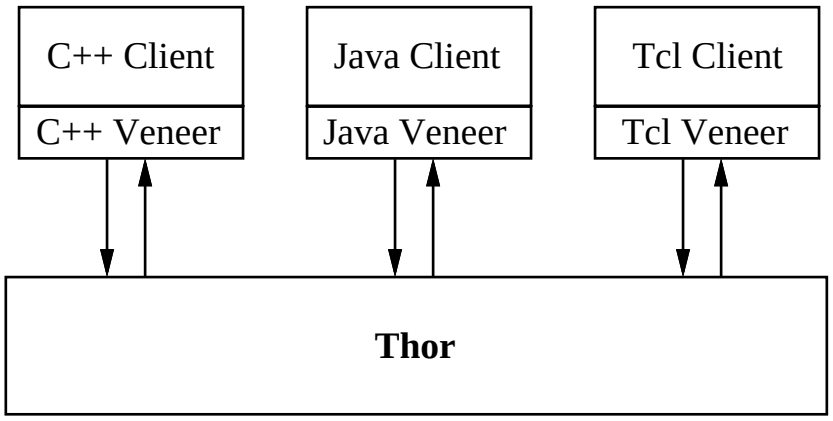

Figure 2-2: Thor Client Interface

2.2 Java

Java is a new programming language from Sun Microsystems. While Java provides a number of interesting features, two are critical to understanding this thesis. First, Java and Theta are very similar, making Java a particularly good match as a client language for Thor. Second, the Java language is designed to be used in systems that use code importing. Java provides a number of language safety features that enable imported code from a remote system to safely run locally.

2.2.1 Java and Theta

One reason that Java is interesting as a client language for Thor is that it closely resembles Theta, the implementation language of Thor objects. Java and Theta are both statically type checked, type-safe, garbage-collected, object-oriented languages with single inheritance. Both languages provide a means of signaling and handling exceptions[LCD+94][Sun95a].

Because of the similarity between the two languages, the client interface between Java and Theta presents Java clients with a natural mapping of Thor's functionality to Java. Thus, a Java client is able to use Java method calls to call Theta methods, and use Java's exception mechanisms to handle Theta exceptions. The Java client programmer interface described in Chapter 3 gives a more detailed description of the mapping between Java and Theta.

2.2.2 Java and Code Importing

One of the most exciting aspects of the Java language is that it is designed to allow code to be stored remotely, sent across a network, and then executed locally. A number of Java language features are designed to make it easy to use Java for code importing. Java code is compiled to bytecodes for an architecture-independent virtual machine, so that the same compiled code can be run on many different platforms. Additionally, a number of security features are built into Java to prevent code that is imported and executed from causing problems to the local machine [JSS95].

Java Language Safety

Java, like C++, has facilities for controlling the access to the variables and methods of objects. These access controls allow objects to be used by non-trusted code with the guarantee that they will not be used improperly. For example, the Java library contains a definition for a File object. The File object has a `public` method (callable by any method) for reading and a low level `private` method (callable only by the object's methods) for reading. The public read method first performs security checks and then calls the private read method. The Java language ensures that non-trusted code can safely manipulate a File object by providing access only to the public methods. Thus, the access control facilities allows programmers to write libraries that are guaranteed by the language to be safe by correctly specifying the library's access controls.

The Java language is also designed to be a type-safe language. This means that the compile-time type and the runtime type of variables are guaranteed to be compatible. This ensures that casts (operations that coerce a run-time type to a given compile-time type) are checked at either compile time or run time to make sure that they are valid. This prevents the forging of access to objects to get around access control. Using our File example from before, this prevents malicious code from casting a File object to its own MyFile type, which

has the same layout as the File type, but with all methods public.

Another safety feature is the elimination of pointers as a data type. This means that pointers cannot be directly manipulated by user code (no pointer arithmetic). This prevents both malicious and accidental misuse of pointers (running off the end of an array for example). Again using our File example, this prevents malicious code from simply accessing the private method directly by using pointer arithmetic starting with the File object's pointer. Clearly this type-safety is a necessary part of the access control facilities of objects, preventing forging.

Code Verification

In the Java system, the Java compiler does not compile a program to machine code; instead it compiles to bytecodes for an architecture-independent virtual machine. The code imported by Thor is transported in its bytecode form. The bytecode does not have the same safety guarantees as the Java language itself. An analysis referred to as *bytecode verification* needs to take place to make sure the imported bytecode follows the safety rules of the Java language.

The Java bytecodes are an instruction set that can be statically type-checked. For example, bytecodes that invoke methods explicitly declare the expected argument and result types. The state in the execution model is stored in a set of local variables and on a stack. The types of each storage location can be determined by straightforward dataflow analysis that infers types for each stack entry and each local variable slot, at each point in the program.

Aside from simple format checks, the bytecode verifier [JSS95][Yel95] checks that the code:

- Does not forge pointers
- Does not violate access restrictions
- Accesses objects according to their types
- Calls methods with appropriate arguments of the appropriate type
- Does not have stack overflows

Chapter 3

Java Client API

The Java client API (Application Programmer Interface), which defines the Java application's interface to Thor, serves two purposes. First, the client API provides the functionality necessary to use Thor from Java in a way that is intuitive for Java programmers. Second, the client API masks the implementation details. The same client API is provided by the two different Java veneer implementations discussed in Chapter 4 and Chapter 5. Since the two implementations provide the same API, the exact same Java client code runs in both versions.

The Java client API is based upon the C++ client API[LDG+96] with some additions to take advantage of features in Java such as exception handling. The client API is divided into two major parts. First, the Java client API provides a set of procedures for supporting basic Thor commands. The second and more substantial part of the Java client API is a library of *stub classes* that represent the Theta types used in Thor.

3.1 Basic Thor Support Commands

The Java client API presents the user with a set of procedures used for basic Thor operations. These procedures provide the functionality of the basic commands in the Thor language-independent interface (see Section 2.1.2) as well as providing a means for converting between Thor types and Java types. The procedures handle all of the client interactions with Thor except for method calls on Thor objects. Table 3.1 gives a summary of the functionality of the provided commands.

Procedure	Functionality
th_init	Initialize the interface between Java and Thor.
th_shutdown	Terminate the interface between Java and Thor.
th_get_root	Returns the *root directory* object from Thor.
th_commit	Commit the current transaction.
th_abort	Abort the current transaction.
th_equal	Compares if two Thor objects are equal.
th_int_to_any th_bool_to_any th_char_to_any	Convert the specified primitive value to an `any` object.
th_force_to_int th_force_to_bool th_force_to_char	Convert an `any` object to the specified primitive value.
string_to_thor_string thor_string_to_string	Converts a Java string to a Thor string and vice versa.

Table 3.1: Java Client Interface Procedures

3.2 Stub Classes

The main design goals of the Java client API is to provide Java programmers with a simple interface to Thor that uses the features of Java. Since Java is an object-oriented programming language, mapping Thor objects and methods to Java objects and methods gives Java programmers a simple and natural interface to Thor. To provide this mapping, the Java client API creates a Java class, called a *stub class*, for each Theta type that is used in Thor.[1] The stub classes are created by using an automatic stub generator. The stub generator uses the Theta type definition to produce a stub class in Java that contains *stub methods* that correspond to methods of the Theta type. For example, the stub generator creates the Java stub class `Th_directory` [2] for the Theta type `directory`. When the Java client refers to Thor objects, it uses *stub objects*, which are instances of the stub classes. The Java client calls Thor methods by invoking the stub method of the stub object. The fourth line of Figure 3-1 illustrates such a method call. Because both Java and Theta use single inheritance, the type hierarchy in Theta is duplicated exactly by the Java stub class hierarchy. For example, in Thor `any` is the supertype of `directory`, so the Java stub class `Th_any` is the superclass of `Th_directory`.

[1] In Theta, a type specifies an interface that is potentially implemented by multiple classes. In Java, the distinction between classes and types is not made. The Java client API provides only a mapping between Theta types and Java classes. The Java API does not distinguish between different Theta classes.

[2] The Java stubs are all in the package `Veneer.stubs`. Thus, the correct class name is `Veneer.stubs.Th_directory`, but for brevity I will omit the full package name when referring to stubs.

```
Veneer.th_init();              /* initialize veneer */
Th_directory root = Veneer.th_get_root();
Th_string s = Veneer.string_to_thor_string("small1");
root.lookup(s);                /* method call to Thor */
Veneer.th_shutdown();          /* shutdown the veneer */
```

Figure 3-1: Java Client code fragment

3.2.1 Method Signatures

Each automatically generated stub class has a Java stub method for each method in the corresponding Theta type. The generated Java stub methods correspond in the number and type of arguments to the Theta methods. Figure 3-2 shows an example Theta method signature, and the corresponding Java method signature.

Theta:

```
lookup(string name) returns (any) signals (not_found)
```

Java:

```
Th_any lookup(Th_string name) throws ThE_not_found
```

Figure 3-2: The signatures of a Theta method and the corresponding Java method

3.2.2 Exceptions

Providing the ability to handle exceptions signaled by Thor using Java's exception capabilities makes it easier for a Java client programmer to use Thor. The Java client API for Thor maps Thor exceptions signaled by calls to Thor methods into Java exceptions. If a Theta method declares that it signals an exception, the corresponding Java method declares that a corresponding Java exception may be thrown. The following code fragment illustrates a simple use of the exception mechanism:

```
try { root.lookup(s); }/* method call to Thor*/
catch (ThE_not_found e) { ... } /* code to handle Thor exception */
```

Unfortunately, the semantics of exceptions differ slightly between Java and Theta. In Java, exceptions are part of the class hierarchy, so a specific exception is actually a full Java object. In Theta, exceptions are simply names for signaled conditions. To handle this difference, the Java client API contains a corresponding exception class for each named Theta exception. All of the Java exceptions that could arise by invoking a stub class method are subclasses of the exception class `ThorException`. This feature is designed to allow the Java client programmer to easily handle only those exceptions that came from Thor by handling the exception type `ThorException`.

3.2.3 Casting

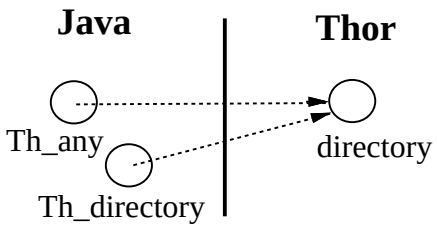

Figure 3-3: Stub Object Mapping

Imprecise Types

The Java client API does not guarantee that the Theta type that a given stub object represents is a precise match with the Java type of that stub object. Instead, as Figure 3-3 illustrates, the type of the Java stub object is guaranteed to be a superclass of the Java class that corresponds to the Thor object's type. In the figure, the Thor object is of type `directory`. As the figure shows, a Java stub object representing the Thor object could be either an instance of `Th_directory` or the more general class `Th_any`.[3] This problem is caused because when returning a Thor object from a stub method the Java client API knows only the return type for the stub methods, and thus must create stub objects of that type. For example, the Java lookup method of Figure 3-2 always returns stub objects of type `Th_any` even if the returned Thor object's type is a subtype of `any`.

Imprecise runtime types create some difficulties when trying to use the Java language's

[3] At first glance it might appear that imprecision of types will cause severe problems because the method dispatch that occurs in Java will not be correct. This is actually not a problem since the real method dispatch occurs inside of Thor.

built-in casting or runtime type discrimination mechanisms with stub objects. In the following code fragment, even if we know that the Thor object represented by `x` is of type `directory`, the cast will not work since the Java stub object's runtime type is `Th_any`:

```
Th_any x = root.lookup(s);
Th_directory d = (Th_directory)x; /* this cast won't work */
```

The same problem occurs when trying to use the Java runtime type discrimination mechanism `instanceof`. Note that this is a problem of false negatives, never false positives; if the cast or `instanceof` operation does work it is guaranteed to be correct.

Coercion Procedures

To solve this problem, the Java client API to Thor provides a procedure for each class that perform a coercion of a Java stub object to a stub object of the given class. The coercion procedures mask the process of first checking the Java runtime type of an object and then querying Thor for the Thor runtime type of the object. These procedures provide a solution that is general enough to encompass both casting and runtime type discrimination by using an exception if the coercion will not work. Using the coercion procedure, the previous example could be written:

```
Th_any x = root.lookup(s);
try { Th_directory d = force_to_Th_directory(x); }
catch (ThE_wrong_type e} { ... } /* if the cast does not work */
```

Chapter 4

RPC Java Veneer

To compare code importing with standard RPC techniques, a Java veneer was implemented using the language-independent interface described in Section 2.1.2. The RPC Java veneer does not take advantage of the safety provided by the Java language because the language-independent interface already provides safety for any client language. Aside from serving as a comparison, the RPC Java veneer implements the Java client API described in Chapter 3, providing remote access to Thor with a client language not previously available.

In Section 4.1, I will first describe some of the implementation details of the the RPC Java veneer.[1] Section 4.2 then discusses an aside about the use of the RPC Java veneer as a simple interface between the World Wide Web and Thor.

4.1 Implementation

The implementation of the RPC Java veneer is based on the preexisting implementation of the C++ veneer for Thor. Both veneers use the language-independent interface described in Section 2.1.2. The RPC Java veneer implementation of the Java client API to Thor has three major parts. First, the RPC Java veneer provides basic primitives that allow Java to communicate with Thor. Second, it provides a mechanism for keeping track of handles for garbage collection purposes as was discussed in Section 2.1.2. Finally, the most significant part of the RPC Java veneer is the implementation of the stub classes that provide the client-side stubs needed by the language-independent interface.

[1]The RPC Java veneer uses an optimization technique called batched futures [BL94], but to simplify the discussion, the use of futures will not be discussed.

4.1.1 Communication Layer

The RPC Java veneer uses the Java TCP/IP socket library for the IPC required to communicate with Thor. On top of the TCP/IP socket layer, the RPC Java veneer uses another layer to provide buffering and some minor bookkeeping required by the Thor-client protocol. A third layer of procedures implements the commands mentioned in Section 3.1. These layers allow the rest of the RPC Java veneer to treat the communication with the server as a black box, simply using a few high level procedures. For example, each of the basic commands listed in Table 3.1 is implemented using this third layer of procedures.

4.1.2 Veneer Handle Table

Section 2.1.2 discussed the fact that for garbage collection purposes, the client must keep track of which handles are still being used. The client must occasionally send to Thor a list of handles that are no longer being used so that Thor can garbage collect the objects referenced by those handles. The RPC Java veneer keeps track of which handles are in use by the Java client by using a reference counting mechanism[BL94]. A veneer handle table is used to keep track of the number of references for each handle seen by the Java client. This table maintains the invariant that the reference count for a given handle is the same as the number of stub objects that refer to that handle. To implement this reference count, when a new Java stub object is created that uses a given handle, the reference count for that handle is incremented in the table. Additionally, when a Java stub object is freed (i.e., collected by the Java garbage collector), the reference count for the handle referred to by the stub object is decremented in the table (see Section 4.1.3). To send Thor the set of free handles, the handle table is occasionally scanned (for example, after every N method calls to Thor), to check for handles with a reference count of zero. The list of free handles is then sent back to Thor, allowing Thor to garbage collect the corresponding objects.

Figure 4-1 illustrates the major features of the veneer handle table. As the figure shows, the Java stub objects store the handle table index (see Section 4.1.3). Thus, the handle 5 has two references since two stub objects store the index 1. Also, the handles 9 and 3 are free since no object stores their handle table index. Thus, the next time the veneer frees handles, the handles 9 and 3 will be sent back to Thor. Finally, note that as was previously mentioned in Section 3.2.3, the same Thor object can be referenced by Java objects that

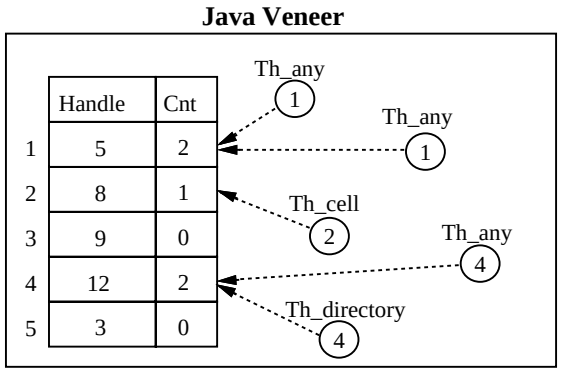

Figure 4-1: Veneer Handle Table

are instances of different Java classes (in this case the two stub objects with index 4).

4.1.3 Stub Classes

The most complex part of the RPC Java veneer is the implementation of the stub classes for the Java client interface to Thor discussed in Section 3.2. The stub classes used by the RPC Java veneer are generated automatically based on Theta type definitions. There were three main issues in the design of the stub classes. First, a representation for the stub objects was chosen. Second, a mechanism was designed for the stub methods. Finally, an implementation of the coercion mechanism described in Section 3.2.3 was designed.

Stub Objects

One straightforward strategy for implementing Java stub objects would have each stub object simply contain the handle of the Thor object it represents. This technique cannot be used because it would not facilitate the RPC veneer's reference counting scheme (described in Section 4.1.2). Instead, each stub object stores an index into the veneer handle table. The creator for each stub class has a handle table index as an argument. The creator then initializes an index variable in the new stub objects and increments the reference count in the handle table.[2]

Java allows the user to associate a *finalizer* procedure with each class[Sun95a]. An object's finalizer runs after the Java garbage collector determines that the object is no

[2]The stub object index, the stub object creation methods, and the handle table itself are accessible only within the veneer package, preventing clients from creating invalid objects or objects with invalid reference counts. In the RPC Java veneer these are only conveniences since subverting the veneer would only crash the client. These techniques become important in Chapter 5.

longer in use. This allows the programmer to associate an action with the destruction of an object. In the RPC Java veneer, when stub objects are destroyed, a reference count needs to be decremented. Thus, stub objects have a finalizer that decrements the reference count associated with the handle table slot of the stub object's index variable. The monitoring of the creation and destruction of all stub objects maintains the simple invariant that the number of objects that have a given handle table index is equal to the reference count in the handle table at that index.

Stub Methods

The RPC Java veneer stub generator creates a Java stub method for each method of a given Theta type. These stub methods act as client-side stubs in the language-independent interface to Thor. Each of the Java methods for a given stub class has an associated *method handle* class variable.[3] When a stub object of that class calls a method, it first checks if the method handle has already been computed. If the method handle has not been computed, before the method call is made a special call to Thor is first used to calculate the method handle. The value of the method handle is then stored so that subsequent calls to that method do not need to recompute the value.

When the stub method for a given stub object is called, it sends Thor the *invoke* command, followed by the object's handle, the method handle, and the method arguments. Since the type of the arguments is known statically (i.e., at stub creation time), the method is hard coded to send its specific arguments. This code uses procedures from the communication layer that take care of mapping Java types (such as `int`, `boolean`, and stub objects) into a form understood by Thor. The stub method then attempts to get a result back from Thor using the communication layer.

The communication layer throws a `ThorException` if the Thor method signals an exception. The `ThorException` object contains a string with the name of the specific Thor exception. If the Java stub method throws a specific exception, the name of the specific exception is compared against each of the known exceptions to determine which exception to throw. This allows the Java client to use Java's own exception handling mechanisms when handling exceptions signaled from Thor. For example, the `directory` method `lookup` signals `not_found`. Thus, the Java stub method `lookup` for `Th_directory` ends with:

[3] A class variable is a variable that is shared by all instances of the class.

```
catch (ThorException e) {
  String m = e.getMessage();
  if (m.equals("not_found")) throw new ThE_not_found();
  throw new Failure("bad Thor exception: lookup"+m);
}
```

If the Thor method returns successfully, the return value is mapped back to the corresponding Java type. If the return value is a handle, the veneer needs to return a stub object of the appropriate type. Unfortunately, when the veneer receives a handle as a return value, it does not get any information about the runtime type of the Thor object represented by the handle. The veneer knows only the declared compile time type (i.e., the method's return type). Thus, the Java stub method will return a new stub object that is an instance of the declared return type for the method. This lack of knowledge about the runtime type of returned objects explains the imprecision of types mentioned in Section 3.2.3. For example, if the return type of a method is `Th_any` (the most general stub type), even if the handle returned corresponds to a `directory` (i.e., a subtype of `any`), a `Th_any` is created with the given handle. To create this object, the veneer first gets the returned handle's index from the handle table, and then creates the new stub with that index.

4.1.4 Implementation of Casting

Each stub class `Th_X` implements a `force_to_Th_X` method that is used for casting as discussed in Section 3.2.3. The `force_to_Th_X` is fairly simple. Given any Java stub object, it should either return an equivalent stub object of the requested type or signal an exception. The basic strategy is to first check if a simple Java coercion would work, and query Thor about the actual types only if necessary (the Thor query is much more expensive). Thus, a check is first made to determine if the runtime Java type is a subtype of the required type. If this check returns `true`, the object itself is returned. Otherwise, a call to the the stub object's Thor method `getClass` is made. This call returns an object of type `Th_Class` that represents the Thor class of the object. Then a call to the `Th_Class` method `subtype` is made to determine if the Thor class is a subtype of the requested class. If this call returns `true`, a new object of the required type is created using the handle table index from

the passed in object. If the subtype call returns `false`, the exception `ThE_wrong_type` is thrown. Unfortunately, this implementation makes casting is a fairly expensive operation, often costing two calls to Thor.

4.2 Applets as Clients

One of the main reasons for the popularity of Java is the ability to run Java programs (known as Applets) over the World Wide Web through browsers such as Netscape Navigator. Java enabled Web browsers load Java code from remote Web servers, and run the Java code on the browser's machine. Since the RPC Java veneer is written entirely in the Java language, programmers can use the veneer in applets. This makes it simple to create Java applets that take advantage of the persistent storage of Thor. Using Java applets over the World Wide Web has the advantage that it gives users platform independent network access to clients that use Thor.

As an example and proof of concept of a Java applet that uses the RPC Java veneer, I created an applet version of the performance benchmarks that were used for the Java veneer (see Chapter 6). Creating the Applet proved to be straightforward. Both the Java veneer code and the client code worked without modification; only additional code for a user interface was written. Previously, Thor had been accessible only on the DEC Alpha. This Applet was subsequently run, without modification, using Netscape Navigator on a Sun Sparcstation, an HP 9000, and a DEC Alpha. Because of the popularity of Java and its ability to make Thor available over the World Wide Web, there are currently plans for creating other Java applets such as a Thor object browser, and an Internet mail client using Thor.

Chapter 5

Code Importing Java Veneer

As discussed in Section 1.2, using code importing in client/server systems has a number of potential advantages. The key advantage over RPC is the potential for improved performance in certain applications. This chapter discusses the issues involved in implementing a Java client interface to Thor that uses code importing to provide good performance while still maintaining server integrity. I describe the key reasons for the performance improvements of code importing and show that by using a safe client language server safety is not threatened.

The chapter is divided into three sections. First, the general architecture of the code-importing Java veneer is described. In Section 5.2 the safety of the code-importing Java veneer will be discussed. Section 5.3 moves on to discuss how the code-importing Java veneer was implemented.

5.1 System Design

The code-importing Java veneer implements the Java client API discussed in Chapter 3, presenting the same interface to Java applications using Thor as the RPC veneer discussed in Chapter 4. The code-importing Java veneer provides two main functions. First, the system allows imported Java code to run in the same address space as the normal Thor server. The ability to run imported Java code is achieved by compiling and linking a slightly modified Java interpreter with the Thor server. The second function that the system provides is a mechanism for calling Thor methods from Java code. The Java interpreter provides *native methods* [Sun95a] for calling routines implemented in C. By using native methods in Java,

a library of C stub routines provides the interface for Java to call Thor operations directly.

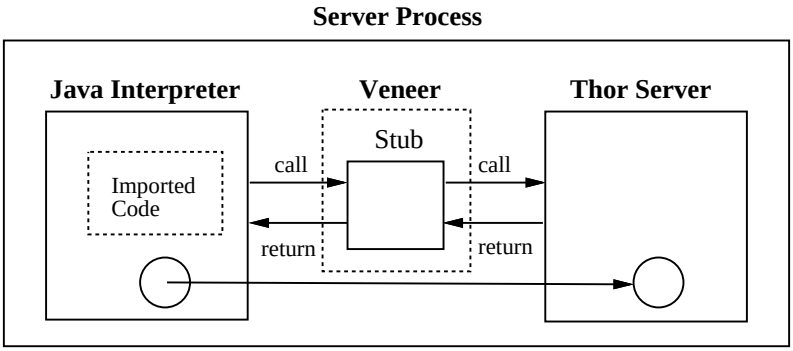

Figure 5-1: Code-Importing Java Veneer

From a performance standpoint, the code-importing veneer design has a number of advantages compared with the RPC veneer. The code importing system uses a simple procedure call to communicate between the client and the server instead of expensive IPC techniques. As figure 5-1 shows, the Java client can also store direct pointers to Thor objects, eliminating an extra indirection. The RPC veneer could not take advantage of the type-safety of the Java language because the language-independent interface does not differentiate between client languages. Since the code-importing Java veneer guarantees that Java is being used as the client language, it can take advantage of Java's type-safety by eliminating runtime type-checks from calls to Thor.

Two other aspects of the code-importing veneer design are important. In normal client/server systems, the client runtime environment cannot be modified. Using code importing, the client's runtime environment can be modified to fit the needs of the server. For example, Section 5.3.2 and Section 5.3.3 show some minor modifications to the embedded Java interpreter that provide significant optimizations for use with the Thor system. The system also provides a narrow, well controlled interface between the client and the server by only allowing the client to interact with the server through the veneer stubs. By providing a narrow interface, it is much easier to reason about properties such as safety.

5.2 Java Safety and Thor

In the RPC veneer, the language-independent interface provides safety by running the Java client and the Thor server in separate address spaces, requiring communication with a

restricted interface, and runtime type checking. In the code-importing Java veneer, the client code runs in the same address space as the server, and calls are not type-checked at run-time. Safety in the code-importing Java veneer is provided by combining the safety properties of the Java language, described in Section 2.2.2, with the system design described in Section 5.1. The use of code verification when the code is first imported guarantees that the imported code follows the safety rules of the Java language.

For the integrity of Thor to be maintained with a Java client running in the same address space, two conditions need to be met. First, the Java client must interact with Thor only through the stub routines provided by the code-importing veneer. Second, the stub routines must be called only with valid arguments that are of the correct types. The first property is satisfied by the safety mechanisms of the Java language. Java cannot directly access memory. Additionally, Java code can only call Java code or C code that specifically provides a Java interface.

The second condition for safety is somewhat more complex. The type-safety of Java guarantees that the stub routines are called with Java objects of the correct type. The more difficult aspect of the second condition is showing that Java stub objects will refer to valid Thor objects with compatible types. By using the access control capabilities of the Java language, the code-importing veneer ensures that stub objects are created only by veneer stub methods, and that the client cannot modify the stub objects. Since the Java stub methods simply call equivalent Thor methods that are type-safe, the stub objects created by the veneer hold valid pointers of the correct type.[1] To prevent client modification of the stub objects, the variable that stores the stub object's pointer to the Thor object it represents is a private variable, which can be accessed only from the veneer's code.[2]

5.3 Implementation

Many of the implementation techniques such as the automatic generation of stubs are similar to techniques used in the RPC veneer. The section focuses on the implementation details that are substantially different in the two veneers. I will first discuss the implementation

[1]The correctness of the garbage collection algorithm used by the code-importing veneer is needed to show that invalidated references are not used by the client. For simplicity, a proof of correctness is omitted.

[2]The code-importing veneer methods are implemented in C, and since they can access private variables they do not have to follow the access restrictions of Java.

of the stub classes, and then give some details about two optimizations used in the code-importing Java veneer.

5.3.1 Stub Classes

As with the RPC veneer, the code-importing Java veneer uses a stub generator to automatically create Java stub classes based on the Theta type definitions. For each stub class, the stub generator for the code-importing Java veneer generates both a Java class that provides the interface for Java code (i.e., method signatures), and a C stub method for each of the type's methods. The types and method signatures provided by the interface for the classes are exactly the same as those of the RPC veneer. The coercion implementation discussed in Section 3.2.3 uses the client API, and thus remains the same in the code-importing Java veneer.

Stub Objects

Unlike the RPC veneer, in the code-importing Java veneer the obvious implementation of stub objects works. A Java stub object stores a pointer to the Thor object it represents. Because the field that holds the pointer is a private member of the stub class, only the veneer code can access or modify that field. Also, unlike the RPC veneer, reference counting is not used in the code-importing Java veneer (see Section 5.3.2), and the stub object creator and destructor do not need to do any special work.

Stub Methods

The automatic stub generator provides a C stub method for each of the stub methods provided by the Java veneer. The stub methods used by the code-importing Java veneer provide the equivalent functionality of both the client and server side stubs in an RPC system. The stub is responsible for marshaling arguments, calling the actual routine, and unmarshaling and returning the result. However, the code-importing veneer stubs are simpler because they do not need to use IPC to communicate and because they do not perform type-checking and validation of arguments.

The implementation of the stub methods in the code-importing Java veneer is straightforward. The Thor object represented by the Java stub object is obtained through a field

access. The arguments to the Java method are first mapped to Thor arguments appropriately (i.e., for other stub objects, the Thor object field is obtained).[3] The stub method then uses a statically known offset to access the appropriate method from the Thor object's method dispatch vector. That Thor method is then called directly. If the Thor method signals an exception (by setting a global variable), the value of the exception is compared against the exceptions the method throws to determine the correct Java exception to create and throw. If the Thor method returns successfully, the return value (if there is one), is mapped back to the appropriate Java type. If the return value is a Thor object, a new stub object of the Java method's return type is created (using a pointer to the returned Thor object) and returned.

5.3.2 Garbage Collection

The RPC Java veneer described in Chapter 4 uses a reference counting mechanism to keep track of Thor objects used by the client, so that the Thor garbage collector could free objects that were once referenced by the client. The integration of the code-importing Java veneer with the Thor system enables a more natural and efficient mechanism to be used. The code-importing Java veneer does not use reference counting; instead it uses the Java garbage collector itself to determine which stub objects are still in use. The code-importing Java veneer uses this technique to tell Thor exactly which objects are still in use by the client, instead of sending a set of objects that are not in use anymore.

To have the veneer keep track of the Thor objects that are being referenced by the Java client, the Java memory allocator and garbage collector were slightly modified.[4] A simple array of veneer stub objects is used to keep track of objects that store direct pointers to Thor objects. This array is used to tell Thor which objects are being referenced by the Java client. When a new stub object is allocated by Java, the stub object is added to the system's array of stub objects. Since stub objects may be freed by the Java garbage collector, the Java garbage collector has been appropriately modified to splice out from the array the freed stub objects at garbage collection time.

[3] The C methods in the Java veneer violate the access restriction on the private variables of the stub objects. But since the C code is trusted, this access violation does not effect the system's safety properties.

[4] The design of the system allowed modifications to the Java interpreter since all Java client code would use the embedded interpreter. This approach would be unreasonable for the RPC Java veneer, since the client code should be able to run with any Java interpreter.

The main modification to the garbage collection mechanism occurs when the Thor garbage collector is called. When the Thor garbage collector is called, a call is first made to the Java garbage collector. When called from Thor, the Java garbage collector performs its normal operation (freeing unused Java objects), with one additional function. Since the Java client stores direct pointers to Thor objects, the Thor garbage collector needs to be told that those objects are still in use. Thus, for each stub object in the array of stub objects, the Java garbage collector tells the Thor garbage collector that the associated Thor object is referenced. After the Java garbage collector finishes, the Thor garbage collector continues its normal function.

Used alone, this garbage collection scheme is only slightly better than the reference counting mechanism used in the RPC Java veneer (because there is no reference counting at stub object creation and destruction). Section 5.3.3 describes another optimization that takes advantage of this garbage collection mechanism as well as making the technique more effective.

5.3.3 A Stub Object Caching Optimization

In both Java veneers the stub methods create a new stub object every time a Thor object is returned by a stub method even if an equivalent stub object already exists. Using this simple implementation for stub methods, the cost of object creation can easily become a dominant cost. For example, with the simple stub method implementation the benchmark discussed in Chapter 6 runs in 14.01 seconds, with over 7 seconds devoted purely to stub object creation. Caching stub objects proved to be an effective optimization, giving an 88 percent hit rate and reducing the benchmark time to 6.7 seconds.

Section 5.3.2 discussed how the code-importing Java veneer keeps track of stub objects for garbage collection purposes using a simple array of stub objects. The stub object caching mechanism is implemented in the code-importing Java veneer by taking advantage of this array of stub objects. Figure 5-2 illustrates the basic idea of the stub object caching implementation. Thor objects store an index into the stub object array if a Java stub object points to that Thor object. Before a stub method creates a new stub object, it can first check to see if the Thor object holds an index. The invariant that needs to be maintained is that for any Thor object `obj` that stores an index n, the stub object at index n of the stub object array refers to `obj`.

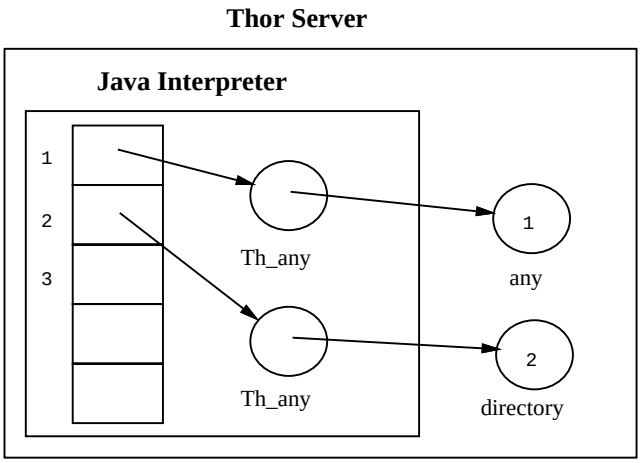

Figure 5-2: The Stub Object Caching Optimization

By storing an index in Thor objects, the Java stub methods can be changed as follows. Instead of creating a new stub object every time a method returns a Java stub object, the method first checks if the Thor object has an index into the stub object array. If the Thor object does not have an index, a new stub object is created, and the index for that stub object is stored in the Thor object. If there is a cached stub object, we would like to return that object. First, a check needs to be made that the cached object is a subtype of the method's return type.[5] If this check passes, the cached object is returned since it will match the return type of the method. Otherwise, a new object of the return type is created and the index for that stub object replaces the old index in the Thor object. This process is shown in Figure 5-3. By using the new index, the Thor object stores a cached stub object that more closely matches its type, and is more likely to be able to be used in future operations. The old object is kept to make sure that the garbage collection algorithm will work as before.

This technique has a number of advantages. First, it eliminates much of the unnecessary object creation that occurs in the Java veneer. Optimally, there would be exactly one stub object for every Thor object that the veneer uses. The original strategy instead created a stub object every time a stub method returned. The improved strategy creates a new stub object only when a method returns a type more specific than any previously seen types for a given object. Another advantage of this technique is that the Java runtime type for a

[5] A subtype check is needed before a cached object is returned because of the type imprecision mentioned in Section 3.2.3 (e.g., the cached object is of type `Th_any`, but the return type is `Th_directory`).

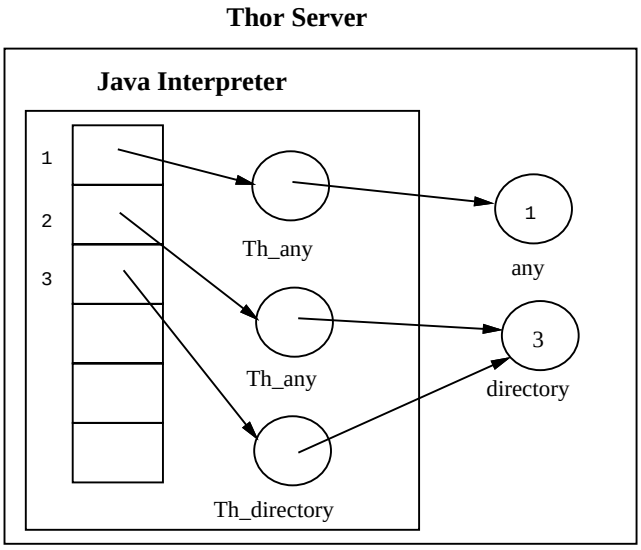

Figure 5-3: Stub Object Caching Optimization: Adding an Object

given Thor object will often more closely approximate the object's true type. For example, suppose a Thor object `dir` is of type `directory`, and the first time it is returned to Java is as type `Th_directory`. Suppose another Java method returns the same object `dir`, but this time as a `Th_any`. Since `dir` already has a cached stub object, the Java stub object of type `Th_directory` will be returned. Thus, this optimization improves the performance of the casting mechanisms discussed in Section 3.2.3, more often allowing the casting operation to use a simple Java cast instead of multiple Thor calls to check the actual Thor type.

Since this object caching optimization works well in the code importing veneer, one might wonder why object caching is not used in the RPC Java veneer. For example, a cached stub object could be stored in the RPC veneer handle table (see Section 4-1) along with the reference count. Unfortunately, keeping a pointer to a stub object will prevent the Java garbage collector from deleting that object. If the object is not deleted, the handle stored by that stub object cannot be freed because the reference count will never be zero.[6] One standard solution that garbage-collected languages use to allow programs to store pointers to objects and still have those objects garbage collected is to have *weak pointers*. A weak pointer is a pointer that the garbage collector will not count as a reference when determining if an object should be garbage collected. Java does not have such a mechanism built into

[6]The reference count for a handle in the RPC Java veneer is the number of stub objects that refer to that handle. It is not a count of the number of pointers to the stub object. Java does not provide the ability to overload assignment and copy operators to keep such a reference count.

the language. The reason that the code-importing Java veneer was able to use the caching optimization was that the modification of the Java runtime allowed the garbage collector to consider its own stub object array as weak pointers. For the RPC veneer, a similar modification to the Java runtime would work. Such a modification would be impractical because the modification would be needed for every Java system that wanted to use the veneer.

Chapter 6

Performance

To evaluate the effectiveness of code importing for client server systems, a number of experiments were designed and run. This chapter describes the results of the experiments and gives some analysis of those results. This chapter is organized as follows. Section 6.1 presents a model that describes the performance in terms of four costs. Section 6.2 outlines the main experiments used to measure performance. Finally, Section 6.2.1 and Section 6.3 provides the experimental results and analysis of the results using the presented performance model.

6.1 Performance Model

The execution time of an application using Thor can be explained by the following model based on a model presented in [LACZ96]. Suppose the client invokes N methods on Thor objects. Each of these calls has an associated cost S. S takes into account the average communication cost, safety costs such as runtime type-checking, and the marshaling and unmarshaling costs. The N method calls to the server use a total of X pairs of domain crossings, each of which has a cost C. The remaining time is divided between the computation performed at the client R_C and the computation performed at the server R_S.[1] The total application running time T can be expressed with the equation:

$$T = X \cdot C + N \cdot S + R_C + R_S \tag{6.1}$$

[1]The model presented in [LACZ96] combines R_C and R_S into a single cost R.

6.2 The Experiments

The intended experiment to compare the performance of code importing against standard RPC techniques involved measuring the performance of the two implementations of the Java veneer. Unfortunately, this experiment provided an unfair comparison since the RPC Java veneer showed extremely poor performance for reasons that were not fundamental to using RPC (see Section 6.2.2). To provide a more meaningful comparison, the code importing Java veneer is compared against a highly optimized C++ veneer based on RPC.

The experiments used were modeled on the experiments used in [LACZ96]. The experiments ran the single-user OO7 benchmark [CDN94]. The OO7 database contains a tree of assembly objects, with leaves pointing to three composite parts chosen randomly from among 500 such objects. Each composite part contains a graph of atomic parts linked by connection objects; each atomic part has 3 outgoing connections. The database used has 20 atomic parts per composite part and a total size of 7 MB.

The results reported are for traversal T1, which performs a depth-first, read-only traversal of the assembly tree and executes an operation on the composite parts referenced by the leaves of this tree. This operation is a depth-first, read-only traversal of the entire graph of a composite part. This traversal was coded in the client language (i.e., Java or C++), using a separate call to Thor for each use of a database object. In total, the T1 traversal requires 450886 call from the client to Thor.

All experiments were run on an isolated DEC 3000/400 workstation with 128 MB of memory using OSF/1 version 3.2 with the client running locally. Unless noted otherwise, the results reported are the average elapsed times obtained in 10 separate trials of each experiment.

The experiments were designed to eliminate costs not associated with computation at the client and server and the communication between them. In particular, the server used a cache large enough to hold the entire database, ensuring that there is no disk I/O. Additionally, the cost of committing the changes performed by the transaction is not included in the measured traversal times.

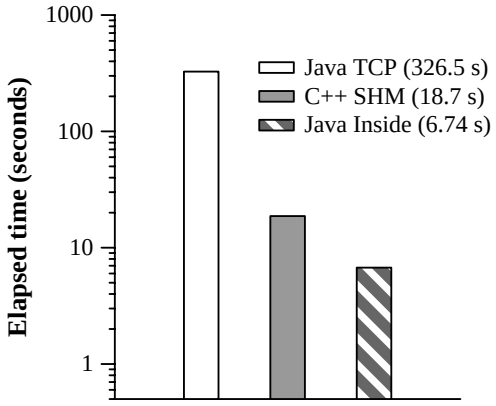

Figure 6-1: OO7 Traversal Comparison

6.2.1 Results

Figure 6-1 shows a comparison between the three different techniques for running the OO7 benchmark.[2] The experiment labeled Java TCP used the Java client with the RPC Java veneer. The experiment labeled C++ SHM used a C++ client with an RPC based C++ veneer that uses a shared memory buffer to communicate with Thor. The experiment labeled Java inside represents the technique presented in Chapter 5. The Java inside experiment used the same Java client as the Java TCP experiment, but used the code-importing Java veneer to provide the interface with Thor. These results show that the code-importing Java veneer performs over 50 times better than the RPC Java veneer and almost 3 times better than the highly optimized RPC C++ veneer. Since the RPC Java veneer performs so poorly, the analysis in Section 6.3 will focus on comparing the RPC Java veneer and the C++ veneer.

6.2.2 Why the RPC Java Veneer is Slow

The results from Figure 6-1 show that the RPC Java veneer performs about 20 times slower than the RPC C++ veneer. The main reason for this difference is that the RPC Java veneer uses TCP/IP to communicate with the server, while the RPC C++ veneer uses a much faster shared memory buffer for communication. Running a PC-sampling profiler on the RPC Java veneer confirmed that the TCP/IP communication was the dominant cost, representing 81% of the traversal time (264 of the 326 seconds). The cost of using TCP/IP

[2]The experiment labeled Java TCP in Figure 6-1 is based on 3 runs (composed of 5 traversals) instead of 10 due to the large amount of time to perform the experiment.

was magnified by the use of an immature implementation of Java that provided poor I/O performance.[3] When the client and server are on the same machine, a version of the RPC Java veneer that used an efficiently implemented shared memory buffer for communication could be used. Such a veneer would have communication characteristics similar to the C++ SHM veneer used.

6.3 Analysis

I have performed analysis and additional experiments to explain the results of Section 6.2.1 in terms of the equation presented in Section 6.1. Table 6.1 and Figure 6-2 present the breakdown of the execution time for the C++ SHM experiment and the Java inside experiment. They show that by eliminating context switching, and severely reducing the communication, safety, and marshaling costs, the code importing veneer outperforms the RPC technique. The table also shows that the performance of the Java client is the major bottleneck in the Java inside experiment. In the future, the use of just-in-time compilation to native machine code should improve the speed of Java. The rest of this section describes how this breakdown was derived.

	C++ SHM	Java Inside
Context Switching ($X \cdot C$)	6.8	0
Communication, Safety and Marshaling ($N \cdot S$)	11.2	.97
Server Compute (R_S)	.7	.59
Client Compute (R_C)	0	5.18
Total	18.7	6.74

Table 6.1: Breakdown of elapsed time for the C++ SHM and Java Inside experiments

Note that for the C++ SHM experiment, the separate server and client computation times were not computed. The .7 second figure represents the combined cost of the client and server computation time. Also note that the cost of code importing and code verification was not included in the traversal time of the Java inside experiment. These costs are discussed in Section 6.3.2.

[3]The implementation of Java that was used is the author's port to the DEC Alpha based on the implementation provided by Sun Microsystems.

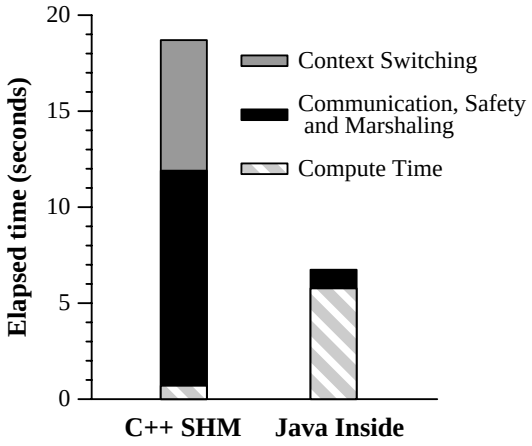

Figure 6-2: Breakdown of elapsed time for the C++ SHM experiments and Java Inside

6.3.1 C++ SHM

The breakdown of the C++ SHM experiment is based on the analysis given in [LACZ96], which provides a more extensive treatment of the performance benefits of different techniques such as batched futures. The C++ SHM experiment presented here uses the RPC based technique that has the best performance. Batched futures are used to reduce the number of context switches and shared memory is used instead of TCP/IP to reduce communications costs. Table 6.1 shows that the dominant cost in the C++ SHM experiment is the overhead of context switching, communication, safety, and marshaling. The actual computation accounts for less than 4 percent of the total running time.

The All-Inside Experiment

To determine the total computation cost $R_C + R_S$, an additional experiment was performed. The all-inside experiment, which took .7 seconds, runs the OO7 traversal completely inside of Thor, using a Theta implementation of the benchmark. The .7 seconds represent only the combined client and server computation time since the all-inside traversal does have any extra costs associated with context switching, communication, safety or marshaling. The .7 seconds is a conservative estimate since the C++ code that performs the client computation (R_C) is faster than the equivalent Theta code that performs the client computation in the all-inside experiment [LACZ96].

The all-inside experiment was optimized with *cord* and *ftoc*, two utilities that reorder procedures in an executable to reduce misses in the code cache. Since these utilities were

not applicable to the code-importing Java veneer, the server compute time calculated for the Java inside experiment does not correspond precisely with the server compute time of the C++ SHM experiment (although they should still be fairly close).

Context Switching

An estimate of the context switching overhead was determined by combining the theoretical cost $X \cdot C$ with data that gave the number of context switches (X) and an experiment that provided the average cost of context switching (C). The number of context switches performed in the C++ SHM experiment was directly calculated to be 138020. By using a version of the C++ SHM veneer that was allowed to (unsafely) run in the same process as the Thor server, the average cost of a pair of context switches was computed to be 49 μs. An estimate for the total context switching cost was thus computed to be 6.8 seconds.

Communication, Safety and Marshaling

The communication, safety and marshaling cost $N \cdot S$ comprised the remaining 11.2 seconds of the total time. The cost $N \cdot S$ includes the costs associated with dynamically type-checking all calls and validating client references as well marshaling and unmarshaling the arguments and return values. The use of batched futures, which improve performance by reducing the number of context switches, increases the communication cost by adding overhead required by the additional machinery that handles batching.

6.3.2 Java Inside Breakdown

The subdivisions provided by the performance model described in Section 6.1 clearly show how code importing can outperform the RPC techniques. The major costs in the RPC experiment are almost entirely eliminated in the Java inside experiment that uses code importing. Since the client code runs inside the server process, there are no context switches. To calculate times for the other three costs, an experiment that gathered data using a PC-sampling profiler was performed while running the Java inside traversal. The profiler provides statistical data that indicates where the program is spending its time by periodically sampling the program counter and recording what procedure is being executed.

The results from the profiler allowed the Java inside experiment to be broken into three major sections corresponding to the three costs of the performance model. The Java inside

experiment spent 76.8 percent of its time directly in the Java interpreter, indicating that the client computation cost R_S is roughly 5.18 seconds. The Java client code communicates with Thor by calling stub procedures that performs the marshaling of arguments and unmarshaling of results. Thus, the cost $N \cdot S$ can be calculated by examining the total time spent in these stub procedures. PC-sampling shows that 14.4 percent of the Java inside traversal is spent in the stub procedures, yielding a total $N \cdot S$ time of .97 seconds.[4] The rest of the time was spent in Thor procedures that perform the server computation. This remaining 8.8 percent yields a server cost R_C of .59 seconds.

Extra Costs of Code Importing

The performance model presented does not fully describe all the overheads of code importing. The results in Figure 6-1 and Table 6.1 do not take into account the overhead of downloading and verifying the Java code. This is because the methodology used in the OO7 benchmarks averages the times of the middle trials of a number of runs to account for cache warming effects. Since the transport and verification overhead of the code importing veneer only occurs in the first run, that overhead is not represented. To determine the cost of verifying the Java code, a simple experiment was run that measured the total time used in verifying client code during the OO7 benchmark. The results show that .084 seconds are spent verifying a total of 5 client classes. To determine the cost of downloading the client code, a simple experiment was run that used TCP/IP to send the 5 client classes between two processes. This experiment showed that transporting the code between processes on the same machine took .007 seconds per file for a total of .035 seconds. Sending the code from a non-local machine would increase this time considerably. But the code importing Java veneer would pay the penalty of a non-local network communication only once, while RPC techniques would be penalized on every call. The total cost of downloading and verification for a local machine was .119 seconds. If this cost were added to the time for a single traversal it would not present an overwhelming cost. Since the downloading and verification was actually amortized over 5 traversals, the cost is even smaller.

[4]The cost of the stub procedures is higher than necessary because Java automatically creates its own stubs for calling native methods. This technique means that two procedure calls are actually made for every call from Java to Thor; first the stub for calling the native method, and then the stub for calling the Thor. By combining the stub generators, the extra procedure call could be eliminated.

Benchmarking Java's Performance

Table 6.1 shows that the time spent in client computation using the Java client was far greater than the time spent in the client computation using C++. To confirm the estimate for the client computation cost of Java and to benchmark Java against C++, I ran an experiment that implements the OO7 traversal with both a Java and C++ program that mimic the Theta code used in the all-inside experiment. These versions of the OO7 traversal did not use Thor, and thus present times that represent only the cost of computation in the given language.

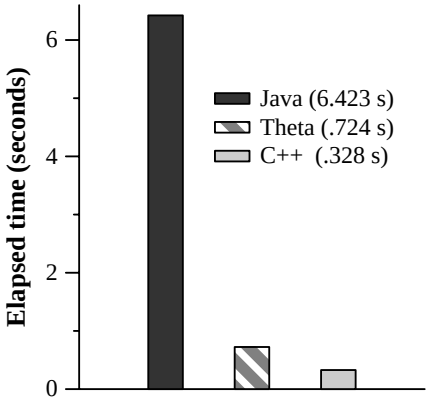

Figure 6-3: Performance comparison of Java against C++

Figure 6-3 shows the comparison between the all-inside traversal that runs in Theta, the traversal written in Java, and a version of the traversal written in C++. In the figure, the difference between the C++ version and the Theta version is due to the extra overhead in Theta of array bounds checking, exception handling, and concurrency control. As the figure shows, this experiment gives a rough confirmation for the computed value of the Java client computation cost. The experiment also indicates that Java is about 20 times slower than C++ for this application. The overwhelming reason for this difference is that the Java experiment is interpreting Java bytecode instead of running native machine code. The use of native machine compilation should improve the performance of Java considerably, making natively compiled Java code competitive with C++ code. Improving the performance of the Java client code should make the code-importing Java veneer more competitive with the all-inside traversal implemented in Theta.

Chapter 7

Related Work

This Chapter presents some comparisons with related work in Thor client optimizations as well as comparisons with work in code importing.

7.1 Thor Client Optimizations

Two recent mechanisms have been presented to reduced communications cost in the Thor Client interface. The first mechanism was presented in Phillip Bogle's thesis on Batched Futures[BL94]. Bogle presents a mechanism for combining (batching) a number of database calls into a single message. The return values of many calls are used only as intermediate value. Combining calls is achieved by deferring such calls, and sending all pending calls when the client needs the actual result of a call. The batched futures technique provides an effective and transparent method for improving performance.

Unfortunately, for most programs, the batching factor (i.e., the number of calls that are sent at once) is fairly small, making the overall gain smaller than is desirable. The use of Batched Control Structures (BCS), presented by Zondervan[Zon95], extends the applicability of call batching. In many programs the batching size using batched futures is limited by conditionals such as those used in loop control structures. BCS allows increased batching size by sending the server additional information about the structure of the program.

The main difference between these techniques and the code importing Java veneer is that they rely on using the same communication technique, but make communications less frequent while the code-importing Java veneer uses a different communication technique (i.e., code importing and then procedure calls). These two mechanisms have some draw-

backs compared with the code-importing Java veneer. First, they trade off performance for exactness of exceptions. If a client program wants to handle a given exception, the call cannot be batched since the exception result is used immediately. In the veneers that use these techniques, this is handled by requiring the client to explicitly request exception information. For client languages that use exceptions (e.g., Java), this technique is unsatisfactory since it does not allow the language's own exception mechanisms to be used. Second, while BCS improves the batching factor, the technique is no longer transparent to the client programmer. BCS requires the use of a slightly modified programming language (i.e., macros are used to replace some control blocks), and requires restructuring of programs that do not use standard control structures. Finally, batched futures requires additional client-side bookkeeping mechanisms in the Thor veneer.

7.2 Code Importing

A number of systems have used code importing for a variety of reasons. For example, the Safe-Tcl language [Bor94] (an extension of the Tcl language) attempts to provide for "Enabled Mail" that would allow users to send email with embedded Safe-Tcl programs. Safe-Tcl uses a variety of techniques to make sure that the language satisfies strong security and portability constraints. Another example is Telescript from General Magic, which also provides code importing in the form of executable content, while claiming to still maintain safety and security[GM]. Currently, a number of World Wide Web browsers use Java to allow users to import code and safely run that code locally. In each of these systems, code importing is used to add functionality to the system. In the code-importing Java veneer, code importing is used for performance reasons.

Importing code has also been shown to be useful in operating systems for improving system performance. The Aegis[EKJ95] operating system allows untrusted code to be downloaded into the kernel. The untrusted code is made safe by using code inspection and sandboxing[WLAG93]. This use of code importing to improve performance is similar to the code-importing Java veneer. The main differences are that verification of Java relies upon type safety of the language, and that once verification is performed the Java code can run without modification. In contrast, sandboxing relies on augmenting the imported code with runtime checks.

The SPIN operating system also allows applications to safely download extensions into the kernel[BSP+95]. The safety of the system is achieved by requiring extensions to be written in a type-safe language. The code-importing Java veneer is similar to SPIN in that it relies upon the type safety of the imported language. In SPIN, the imported code uses the same language as the SPIN system language. Unlike SPIN, the Java interface to Thor is importing code from a language (Java) that is different from the language of Thor itself (Theta). Additionally, the imported Java code is already compiled and must be separately verified.

Chapter 8

Conclusion

This first part of this chapter presents a summary of the work done in this thesis as well as some conclusions that can be drawn from that work. Section 8.2 concludes with some suggestions for future work.

8.1 Summary

This thesis has presented a comparison of two techniques for safe client/server communications. First, a Java veneer for Thor was implemented using the standard safety technique of isolating the client from the server. After the baseline veneer was implemented, a Java veneer for Thor was designed and implemented that used code importing of a type-safe language combined with code verification to maintain server integrity. This technique proved to be very effective, allowing a number of optimizations that provided much higher performance. I expect that the use of code importing in other client/server systems will show similar performance increases while preserving server integrity.

8.2 Future Work

There are a number of different directions for future work that could be explored. The most obvious extension to the code-importing Java veneer would be to use compilation techniques to improve performance. Compiling Java to native machine code should probably increase speed by about a factor 10. Since the bottleneck in the code importing Java veneer is the speed of the Java client, compilation should provide considerable performance

improvements. Many Java developers are exploring the idea of just-in-time (JIT) compilation. With this technique, key pieces of the Java code are compiled as needed, after the code has been imported and verified to be safe. JIT compilation would clearly be applicable to the code-importing Java veneer.

Adding the ability to use the code-importing Java veneer non-locally is another straightforward improvement. Currently, no mechanism has been provided to allow remote clients to connect to Thor and send Java client code to run. A simple version of this extension merely requires a protocol to be implemented for the client to initiate and send Java code, as well as a mechanism for Thor to request any other Java classes that are needed to run the imported client. This simple extension would allow clients to use the code-importing Java veneer non-locally only in a very coarse manner. In other words, the required computation needs to be completely expressed in one full Java program. A more complex area for future work is the design of a mechanism to allow Java clients to use the performance advantages of the code-importing Java veneer in a more fine grained manner. For example, the client may want to send code to Thor that is not self contained (i.e., that relies upon local state).

A number of interesting, but fairly involved extensions involve integrating Java more closely with Thor. For example, Thor could be extended to store Java objects as well as Theta objects. Another example is that Thor could use Java bytecodes as a stored code standard. Because of the popularity of Java, it is likely that many languages will develop compiler backends to produce Java bytecode. Using Java bytecode as a code standard for Thor might allow those languages to easily use Thor.

Bibliography

[BL94] Philip Bogle and Barbara Liskov. Reducing cross-domain call overhead using batched futures. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1994.

[Bor94] Nathaniel S. Borenstein. Email with a mind of its own: The safe-tcl language for enabled mail. In *ULPAA*, 1994.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the spin operating system. In *Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[CDN94] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. Technical Report; Revised Version dated 7/21/1994 1140, University of Wisconsin-Madison, 1994. WWW users: see URL ftp://ftp.cs.wisc.edu/OO7.

[EKJ95] Dawson R. Engler, Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[GM] Inc. General Magic. An introduction to safety and security in telescript. Available at `http://cnn.genmagic.com/Telescript/TDE/security.html`.

[JSS95] Hotjava(tm): The security story, 1995. Available at `http://java.sun.com/1.0alpha3/doc/security/security.html`.

[LAC+96] Barbar Liskov, Atul Adya, Miguel Castro, Mark Day, Sanjay Ghemawat, Robert Gruber, Umesh Maheshwari, Andrew C. Myers, and Liuba Shrira. Safe and efficient sharing of persistant objects in Thor. In *ACM SIGMOD Int. Conf. on Management of Data*, 1996.

[LACZ96] B. Liskov, A. Adya, M. Castro, and Q. Zondervan. Type-safe heterogeneous sharing can be fast. In *Seventh International Workshop on Persistent Object Systems*, May 1996.

[LCD+94] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawhat, Robert Gruber, Paul Johnson, and Andrew C. Myers. Theta reference manual. Technical Report 88, Laboratory for Computer Science, MIT, February 1994. Also available at `http://www.pmg.lcs.mit.edu/papers/thetaref/`.

[LDG+96] Barbar Liskov, Mark Day, Sanjay Ghemawat, Robert Gruber, Umesh Maheshwari, Andrew C. Myers, and Liuba Shrira. The language-independent interface

of the Thor persistent object system. In *Object-Oriented Multi-Database Systems*, pages 570–588. Prentice Hall, 1996.

[Sun95a] Sun Microsystems. *Java Language Specification*, version 1.0 beta edition, October 1995. Available at `http://ftp.javasoft.com/docs/vmspec.ps.Z`.

[Sun95b] Sun Microsystems. *The Java Virtual Machine Specification*, release 1.0 beta edition, August 1995. Available at `http://ftp.javasoft.com/docs/javaspec.ps.tar.Z`.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *14th ACM Symp. on Operating System Principles*, pages 203–216, Asheville, NC, December 1993.

[Yel95] Frank Yellin. Low-level security in Java, December 1995. Presented at the Fourth International World Wide Web Conference, Dec. 1995.

[Zon95] Quinton Y. Zondervan. Increasing cross-domain call batching using promises and batched control structures. Technical Report MIT/LCS/TR-658, Laboratory for Computer Science, MIT, Cambridge, MA, June 1995. Master's thesis.