February 28, 1994
Revised December 4, 1995

SRC Research Report

115

Network Objects

Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

Network Objects

Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber

February 28, 1994
Revised December 4, 1995

Affiliations

Susan Owicki is an independent consultant. Her current address is 956 N. California Ave, Palo Alto, CA 94303, U.S.A. (e-mail: owicki@mojave.stanford.edu).

Publication History

An earlier version of this report appeared in the Proceedings of the Fourteenth ACM Symposium on Operating System Principles, Asheville, North Carolina, December 5-8, 1993.

This report is to appear in Software – Practice & Experience, John Wiley & Sons, Ltd.

Authors' Abstract

A network object is an object whose methods can be invoked over a network. The Modula-3 network objects system is novel for its overall simplicity. It provides distributed type safety through the narrowest surrogate rule, which allows programmers to export new versions of distributed services as subtypes of previous versions. This report describes the design and implementation of the system, including a thorough description of realistic marshaling algorithms for network objects, precise informal specifications of the major system interfaces, lessons learned from using the system, and performance results.

Contents

1 Overview

In pure object-oriented programming, clients cannot access the concrete state of an object directly, but only via the object's methods. This methodology applies beautifully to distributed computing, since the method calls are a convenient place to insert the communication required by the distributed system. Systems based on this observation began to appear about a decade ago, including Argus[16], Eden[1], and early work of Shapiro[24], and more keep arriving every day. It seems to be the destiny of distributed programming to become object-oriented, but the details of the transformation are hazy. Should objects be mobile or stationary? Should they be communicated by copying or by reference? Should they be active? Persistent? Replicated? Is the typical object a menu button or an X server? Is there any difference between inter-program typechecking and intra-program typechecking?

We believe that the way to make progress in these issues is to discuss them in the context of real implementations. To this end, this report describes a distributed programming system for Modula-3, that we call *network objects*.

Network objects provide functionality similar to remote procedure call (RPC), but they are more general and easier to use. Our network objects are not mobile, but we make it easy to communicate objects either by copying or by reference. Our objects are passive: they have no implicitly associated thread of control, nor is there any implicit synchronization associated with calling their methods. Our objects are not persistent or replicated. They are sufficiently lightweight that it would be perfectly practical to use one per menu button. We provide strong inter-program typechecking.

The primary distinguishing aspect of our system is its simplicity. We restricted our feature set to those features that we believe are valuable to all distributed applications (distributed type-checking, transparent invocation, powerful marshaling, efficient and convenient access to streams, and distributed garbage collection), and we omitted more complex or speculative features (object mobility, transactions).

We organized the implementation around a small number of quite simple interfaces, each of which is described in this report. The report also describes a number of implementation details that have been omitted from previously published work, including simple algorithms for marshaling and unmarshaling network objects in a heterogeneous network. All of this material makes the report longer than we would like, but to make progress in the complicated design space of distributed object systems it seems necessary to describe real systems in more detail than is customary in the research literature.

We now briefly introduce some of the central aspects of our design.

Distributed typechecking. Our system provides strong typechecking via the *narrowest surrogate rule*, which will be described in detail below. In a distributed environment it can be very difficult to release a new version of a service, since old clients must be supported as well as new ones. The narrowest surrogate rule allows a programmer to release a new version of the service as a subtype of the old version, which supports both old and new clients and ensures type safety. The narrowest surrogate rule could also be useful in other situations where separately compiled programs need to communicate in a type-safe way; for example, it could be used to solve the problem of version skew in shared libraries.

Transparent remote invocation. In our system, remote invocations are syntactically identical to local ones; their method signatures are identical. A client invoking a method of an object need not know whether the object is local or remote.

Powerful marshaling. As in any distributed programming system, argument values and results are communicated by *marshaling* them into a sequence of bytes, transmitting the bytes from one program to the other, and then unmarshaling them into values in the receiving program. The marshaling code is contained in stub modules that are generated from the object type declaration by a stub generator. Our marshaling code relies heavily on a general-purpose mechanism called *pickles*. Pickles use the same runtime-type data structures used by the local garbage collector to perform efficient and compact marshaling of arbitrarily complicated data types. Our stub generator produces in-line code for simple types, but calls the pickle package for complicated types. This combination strategy makes simple calls fast, handles arbitrary data structures, and guarantees small stub modules.

We believe it is better to provide powerful marshaling than object mobility. The two facilities are similar, since both of them allow the programmer the option of communicating objects by reference or by copying. Either facility can be used to distribute data and computation as needed by applications. Object mobility offers slightly more flexibility, since the same object can be either sent by reference or moved; while with our system, network objects are always sent by reference and other objects are always sent by copying. However, this extra flexibility doesn't seem to us to be worth the substantial increase in complexity of mobile objects. For example, a system like Hermes[5], though designed for mobile objects, could be implemented straightforwardly with our mechanisms.

Leveraging general-purpose streams. Our whole design makes heavy use of object-oriented *buffered streams*. These are abstract types representing buffered streams in which the method for filling the buffer (in the case of input streams) or flushing the buffer (in the case of output streams) can be overridden differently in different subtypes. The representation of the buffer and the protocol for invoking the flushing and filling methods are common to all subtypes, so that generic facilities can deal with buffered streams efficiently, independently of where the bytes are coming from or going to. To our knowledge these streams were first invented by the designers of the OS6 operating system[26]. In Modula-3 they are called *readers* and *writers*[18].

Because we use readers and writers, our interface between stubs and protocol-specific communication code (which we call *transports*) is quite simple. This choice was initially controversial, and viewed as a likely source of performance problems. However, since readers and writers are *buffered* streams, it is still possible for the stubs to operate directly on the buffer when marshaling and unmarshaling simple types, so there is not much loss of efficiency for simple calls. And for arguments that need to be pickled, it is a further simplification that the streams used by the transport interface are of the same type as those assumed by the pickle package.

Marshaling support for streams. Inter-process byte streams are more convenient and efficient than RPC for transferring large amounts of unstructured data, as critics have often pointed out. We have therefore provided special marshaling support for Modula-3's standard stream types (readers and writers). We marshal readers and writers by defining surrogate readers and writers as subtypes of the abstract stream types. To communicate a stream from one program to another, a surrogate stream is created in the receiving program. Data is copied over the network between the buffers of the real stream and the surrogate stream.

Here again the prevalence of readers and writers is important: the stream types that the marshaling code supports are not some new kind of stream invented for marshaling purposes, but exactly the readers and writers used by the existing public interfaces in the Modula-3 library.

Distributed garbage collection. Garbage collection is a valuable tool for programming distributed systems, for all the reasons that apply to programs that run in a single address space. Our distributed collector allows network objects to be marshaled freely between processes without fear of memory leakage. We employ a fault-tolerant and efficient algorithm for distributed garbage collection that is a generalization of reference counting; it maintains a set of identifiers for processes with references to an object. Distributed collection is driven by the local collectors in each process; there is no need for global synchronization. Our algorithm, however, does not collect circular structures that span more than one address space.

1.1 Related work

We have built closely on the ideas of Emerald[14] and SOS[25]; our main contribution has been to select and simplify the essential features of these systems. One important simplification is that our network objects are not mobile. Systems like Orca[2] and Amber[7] aim at using objects to obtain performance improvements on a multiprocessor. We hope that our design can be used in this way, but our main goal was to provide reliable distributed services, and consequently our system is quite different. For example, the implementations of Orca and Amber described in the literature require more homogeneity than we can assume. (Rustan Leino has implemented a version of Modula-3 network objects on the Caltech Mosaic, a fine-grained mesh multiprocessor[15]. But we will not describe his work here.)

Systems like Argus[16, 17] and Arjuna[8] are like network objects in that they aim to support the programming of reliable distributed services. However, they differ from network objects by providing larger building blocks, such as stable state and multi-machine atomic transactions, and are oriented to objects that are implemented by whole address spaces. Our network objects are more primitive and fine-grained.

The Spring *subcontract* is an intermediary between a distributed application and the underlying object runtime[11]. For example, switching the subcontract can control whether objects are replicated. A derivative of this idea has been incorporated into the *object adaptor* of the Common Object Request Broker Architecture[19]. We haven't aimed at such a flexible structure, although our highly modular structure allows playing some similar tricks, for example by building custom transports.

1.2 Definitions

A Modula-3 *object* is a reference to a data record paired with a method suite. The method suite is a record of procedures that accept the object itself as a first parameter. A new object type can be defined as a *subtype* of an existing type, in which case objects of the new type have all the methods of the old type, and possibly new ones as well (inheritance). The subtype can also provide new implementations for selected methods of the supertype (overriding). Modula-3 objects are always references, and multiple inheritance is not supported. A Modula-3 object includes a typecode that can be tested to determine its type dynamically[18].

A *network object* is an object whose methods can be invoked by other programs, in addition to the program that allocated the object. The program invoking the method is called the *client* and the program containing the network object is called the *owner*. The client and owner can be running on different machines or in different address spaces on the same machine.

A remote reference in a client program actually points to a local object whose methods perform remote procedure calls to the owner, where the corresponding method of the owner's object is invoked. The client program need not know whether the method invocation is local or remote. The local object is called a *surrogate* (also known as a proxy).

The surrogate object's type will be declared by a stub generator rather than written by hand. This type declaration includes the method overrides that are analogous to a conventional client stub module. There are three object types to keep in mind: the network object type `T` at which the stub generator is pointed; the surrogate type `TSrg` produced by the stub generator, which is a subtype of `T` with method overrides that perform RPC calls; and the type `TImpl` of the real object allocated in the owner, also a subtype of `T`. The type `T` is required to be a *pure* object type; that is, it declares methods only, no data fields. The type `TImpl` generally extends `T` with appropriate data fields.

If program `A` has a reference to a network object owned by program `B`, then `A` can pass the reference to a third program `C`, after which `C` can call the methods of the object, just as if it had obtained the reference directly from the owner `B`. This is called a *third party transfer* . In most conventional RPC systems, third party transfers are problematical; with network objects they work transparently, as we shall see.

For example, if a network node offers many services, instead of running all the servers it may run a daemon that accepts a request and starts the appropriate server. Some RPC systems have special semantics to support this arrangement, but third-party transfers are all that is needed: the daemon can return to the client an object owned by the server it has started; subsequent calls by the client will be executed in the server.

When a client first receives a reference to a given network object, either from the owner or from a third party, an appropriate surrogate is created by the unmarshaling code. Care is required on several counts.

First, different nodes in the network may use different underlying communications methods (transports). To create the surrogate, the code in the client must select a transport that is shared by the client and owner—and this selection must be made in the client before it has communicated with the owner.

Second, the type of the surrogate must be selected. That is, we must determine the type `TSrg` corresponding to the type `TImpl` of the real object in the owner. But there can be more than one possible surrogate type available in the client, since `TSrg` is not uniquely determined by `TImpl`. As we shall see, this situation arises quite commonly when new versions of network interfaces are released. The ambiguity is resolved by the *narrowest surrogate rule*: the surrogate will have the most specific type of all surrogate types that are consistent with the type of the object in the owner and for which stubs are available in the client and in the owner. This rule is unambiguous because Modula-3 has single inheritance only.

Since the type of the surrogate depends on what stubs have been registered in the owner as well as in the client, it can't be determined statically. A runtime type test will almost always be necessary after the surrogate is created.

1.3 Examples

The narrowest surrogate rule is useful when network interfaces change over time, as they always do. This section presents some examples to illustrate this utility. The examples also show

how network objects generalize and simplify features of conventional RPC. The examples are based on the following trivial interface to a file service:

```
INTERFACE FS;

IMPORT NetObj;

TYPE
  File = NetObj.T OBJECT METHODS
    getChar(): CHAR;
    eof(): BOOLEAN
  END;
  Server = NetObj.T OBJECT METHODS
    open(name: TEXT): File
  END;

END FS.
```

The interface above is written in Modula-3. It declares object types `FS.File`, a subtype of `NetObj.T` extended with two methods, and `FS.Server`, a subtype of `NetObj.T` with one extra method. Any data fields would go between `OBJECT` and `METHODS`, but these types are pure. It is conventional to name the principal type in an interface `T`; thus `NetObj.T` is the principal type in the `NetObj` interface.

In our design, all network objects are subtypes of the type `NetObj.T`. Thus the interface above defines two network object types, one for opening files, the other for reading them. If the stub generator is pointed at the interface `FS`, it produces a module containing client and server stubs for both types.

Here is a sketch of an implementation of the `FS` interface:

```
MODULE Server EXPORTS Main;

IMPORT NetObj, FS;

TYPE
  File = FS.File OBJECT
    <buffers, etc.>
  OVERRIDES
    getChar := GetChar;
    eof := Eof
  END;
  Svr = FS.Server OBJECT
    <directory cache, etc.>
  OVERRIDES
    open := Open
  END;

<code for GetChar, Eof, and Open>;

BEGIN
  NetObj.Export(NEW(Svr), "FS1");
  <pause indefinitely>
END Server.
```

The call `NetObj.Export(obj, nm)` *exports* the network object `obj`; that is, it places a reference to it in a table under the name `nm`, whence clients can retrieve it. The table is typically contained in an *agent* process running on the same machine as the server.

Here is a client, which assumes that the server is running on a machine named `server`:

```
MODULE Client EXPORTS Main;
  IMPORT NetObj, FS, IO;

VAR
  s: FS.Server := NetObj.Import("FS1", NetObj.Locate("server"));
  f := s.open("/usr/dict/words");

BEGIN
  WHILE NOT f.eof() DO IO.PutChar(f.getChar()); END
END Client.
```

The call `NetObj.Locate(nm)` returns a handle on the agent process running on the machine named `nm`. The call to `NetObj.Import` returns the network object stored in the agent's table under the name `FS1`; in our example this will be the `Svr` object exported by the server. `Import`, `Export`, and `Locate` are described further in the section below on bootstrapping.

The client program invokes the remote methods `s.open`, `f.getChar`, and `f.eof`. The network object `s` was exported by name, using the agent running on the machine `server`. But the object `f` is anonymous; that is, it is not present in any agent table. The vast majority of network objects are anonymous; only those representing major services are named.

For comparison, here is the same functionality as it would be implemented with an RPC that is not object-oriented, such as DCE RPC[20]. The interface would define a file as an *opaque type*:

```
INTERFACE FS;

TYPE T;

PROC Open(n: TEXT): T;
PROC GetChar(f: T): CHAR;
PROC Eof(f: T): BOOL;

END FS.
```

A conventional RPC stub generator would transform this interface into a client stub, a server stub, and a modified client interface containing explicit binding handles:

```
INTERFACE FSClient;

IMPORT FS;

TYPE Binding;

PROCEDURE Import(hostName: TEXT): Binding;
PROCEDURE Open(b: Binding, n: TEXT): FS.T;
PROCEDURE GetChar(b: Binding, f: FS.T): CHAR;
PROCEDURE Eof(b: Binding, f: FS.T): BOOL;

END FSClient.
```

The server would implement the `FS` interface and the client would use the `FSClient` interface. In `FSClient`, the type `Binding` represents a handle on a server exporting the `FS` interface, and the type `T` represents a so-called *context handle* on an open file in one of these servers. Here is the same client computation coded using the conventional version:

```
MODULE Client;

IMPORT FSClient, IO;

VAR
  b := FSClient.Import("server");
  f := FSClient.Open(b, "/usr/dict/words");
BEGIN
  WHILE NOT FSClient.Eof(b, f) DO
    IO.PutChar(FSClient.GetChar(b, f))
  END
END Client.
```

Comparing the two versions, we see that the network object `s` plays the role of the binding `b`, and the network object `f` plays the role of the context handle `f`. Network objects subsume the two notions of binding and context handle.

In the conventional version, the signatures of the procedures in `FSClient` differ from those in `FS`, because the binding must be passed. Thus the signature is different for local and remote calls. (In this example, DCE RPC could infer the binding from the context handle, allowing the signatures to be preserved; but the DCE programmer must be aware of both notions.) Moreover, although conventional systems tend to allow bindings to be communicated freely, they don't do the same for context handles: It is an error (which the system must detect) to pass a context handle to any server but the one that created it.

The conventional version becomes even more awkward when the same address space is both a client and a server of the same interface. In our `FS` example, for example, a server address space must instantiate the opaque type `FS.T` to a concrete type containing the buffers and other data representing an open file. On the other hand, a client address space must instantiate the opaque type `FS.T` to a concrete type representing a context handle. (This type is declared in the client stub module.) These conflicting requirements make it difficult for a single address space to be both a client and a server of the same interface. This problem is called *type clash*. It can be finessed by compromising on type safety; but the network object solution avoids the problem neatly and safely.

Object subtyping together with the narrowest surrogate rule make it easy to ship a new version of the server that supports both old and new clients, at least in the common case in which the only changes are to add additional methods. For example, suppose that we want to ship a new file server in which the files have a new method called `close`. First, we define the new type as an extension of the old type:

```
TYPE
  NewFS.File = FS.File OBJECT METHODS
    close()
  END;
```

Since an object of type `NewFS.File` includes all the methods of an `FS.File`, the stub for a `NewFS.File` is also a stub for an `FS.File`. When a new client—that is, a client linked with stubs for the new type—opens a file, it will get a surrogate of type `NewFS.File`, and be able to invoke its `close` method. When an old client opens a file, it will get a surrogate of type `FS.File`, and will be able to invoke only its `getChar` and `eof` methods. A new client dealing with an old server must do a runtime type test to check the type of its surrogate.

The extreme case of the narrowest surrogate rule occurs when a network object is imported into a program that has no stubs linked into it at all. In this case the surrogate will have type `NetObj.T`, since every program automatically gets (empty) stubs for this type. You might think that a surrogate of type `NetObj.T` is useless, since it has no methods. But the surrogate can be passed on to another program, where its type can become more specific. For example, the agent process that implements `NetObj.Import` and `NetObj.Export` is a trivial one-page program containing a table of objects of type `NetObj.T`. The agent needs no information about the actual subtypes of these objects, since it doesn't call their methods, it only passes them to third parties.

1.4 Failure semantics and alerts

An ordinary procedure call has no special provision for notifying the caller that the callee has crashed, since the caller and the callee are the same program. But a remote procedure call mechanism must define some *failure semantics* that cover this situation, in order to make it possible to program reliable applications.

In theory, distributed computations can be more reliable than centralized ones, since if a machine crashes, the program can shift the computation to use other machines that are still working. But it isn't easy to put this theory into practice. Many distributed systems end up being *less* reliable than their centralized equivalents, because they are vulnerable to the failure of many machines instead of just one. Leslie Lamport, prominent both as a theorist and as a suffering user, has facetiously defined a distributed system as one in which "the failure of a computer you didn't even know existed can render your own computer unusable".

Many methodologies and tools have been proposed to aid in programming replicated distributed services that survive the failures of individual replicas. Our network object system is intended to provide a more fundamental communications primitive: replicated services can be built out of network objects, but so can non-replicated services.

The failure semantics of network objects are similar to those of many conventional RPC systems. The runtime raises the exception `NetObj.Error` in the client if the owner crashes while the method call is in progress. Therefore, in serious applications, all methods of network objects should include this exception in their `RAISES` set. (Failure to include the exception would cause the client to crash in this situation, which is usually not what you want a serious application to do.)

Unfortunately, there is no absolutely reliable way that one machine can tell if another has crashed, since the communication network can fail, and a live machine can't distinguish itself from a dead machine if it cannot communicate. Therefore, the exception `NetObj.Error` doesn't guarantee that the owner has crashed: possibly communication has failed. In the latter case, the method call in the owner may continue to execute, even while the client runtime raises `NetObj.Error`. The abandoned computation in the owner is called an *orphan*. To build an application that is robust in the presence of communication failures, the programmer must ensure that the computation meets its specification even in the presence of orphaned computations.

Modula-3 provides a mechanism for *alerting* a thread. This is not an interrupt, but a polite request for the thread to stop at the next convenient point and raise a pre-defined exception. When programming a lengthy computation that might for any reason be subject to cancellation, it is good style to check periodically to see if the thread has been alerted.

If a thread engaged in a remote call is alerted, the runtime raises `NetObj.Error` in the calling thread and simultaneously attempts to notify and alert the server thread executing the call. The reason that `NetObj.Error` is raised is that there is no guarantee that the attempt to alert the server thread will succeed; therefore, an orphan may have been created.

The network object system also uses alerts to handle the situation in which a client crashes while it has an outstanding remote method call. In this case, the network object runtime alerts the thread that is executing the method call in the owner. Therefore, most methods of network objects should include `Thread.Alerted` in their `RAISES` sets.

2 Implementation

This subsection describes the structure of our implementation. Much of the lower levels of our system are similar to that of conventional RPC, as described by Birrell and Nelson[4]. We will concentrate on the implementation aspects that are new in network objects.

2.1 Assumptions

We implemented our system with Modula-3 and Unix, but our design would work on any system that provides threads, garbage collection, and object types with single inheritance. At the next level of detail, we need the following capabilities of the underlying system:

1. Object types with single inheritance.
2. Threads (lightweight processes).
3. Some form of reliable, inter-address-space communication.
4. Garbage collection, together with a hook for registering a cleanup routine for selected objects to be called when they are collected (or explicitly freed).
5. Object-oriented buffered streams (readers and writers).
6. Runtime type support as follows. Given an object, to determine its type; given a type: to determine its supertype, to allocate and object of the type, and to enumerate the sizes and types of the fields of the type.
7. A method of communicating object types from one address space to another.

We will elaborate on the last item.

The Modula-3 compiler and linker generate numerical typecodes that are unique within a given address space. But they are not unique across address spaces and therefore cannot be used to communicate types between address spaces. Therefore, the Modula-3 compiler computes a *fingerprint* for every object type appearing in the program being compiled. A fingerprint is a sixty-four bit checksum with the property that (with overwhelming probability) two types have the same fingerprint only if they are structurally identical. Thus a fingerprint denotes a type independently of any address space. Every address space contains two tables mapping between its typecodes and the equivalent fingerprint. To communicate a typecode from one address space to another, the typecode is converted into the corresponding fingerprint in the sending address space and the fingerprint is converted into the corresponding typecode in the receiving address space. If the receiving program does not contain a code for the type being sent, then the second table lookup will fail.

2.2 Pickles

We use a mechanism known as *pickles* to handle the more complex cases of marshaling, specifically those that involve references types as arguments or results. Our pickles package is similar to the value transmission mechanism described by Herlihy and Liskov[12], who seem to be the first to have described the problem in detail. However, our package is more general because it handles subtyping and dynamic types.

For simple usage, our pickle package provides a simple interface. `Pickle.Write(ref, wr)` writes a flattened representation of the dynamically typed value `ref` to the writer `wr`. `Pickle.Read(rd)` reads the representation of a value from the reader `rd` and returns a dynamically typed reference to a copy of the original value.

The pickle package relies on the compile-time and runtime type support described earlier, and in particular on the existence of type fingerprints. Given this support, the basic method for writing a pickle is quite simple. It writes the fingerprint of the given value's type on the byte stream, followed by the referent's data fields. The method recurses on any constituent values that are themselves references types. Reading is the inverse operation: read a fingerprint, allocate a value, examine the type, read data fields and recursively read reference fields.

One minor complication is the problem of keeping the values independent of machine architecture (for example, byte order or word length). We do this by encoding the salient properties of the architecture in a small number of bytes at the start of the pickle, then writing the pickle in the sender's native form. This approach is efficient in homogeneous cases, and no more costly than anything else in heterogeneous cases. We assume that all architectures can be described by our header. If there were an aberrant architecture, its pickle package would be required to map to and from a standard one on sending and receiving.

A slightly more significant complication is detecting and dealing with multiple occurrences of the same reference within a single pickled value. This happens in cyclic structures and also in graph-like structures that are not trees. (We make no attempt to preserve sharing between separate pickles.)

When writing a pickle, the sender maintains a hash table keyed by references. The values in this table are small integers, allocated sequentially within each particular pickle. When a reference is first encountered in writing a pickle, it is entered in the table and allocated a small integer. This integer is written on the byte stream after the reference's fingerprint, as the defining occurrence. Then the pickle package writes the referent by recursing. If a reference is encountered for a second or subsequent time in a single pickle, the reference's small integer is found in the hash table and written on the byte stream as a subsequent occurrence; in this case there is no need to examine or write the referent.

When reading a pickle, the receiver maintains an array indexed by these small integers. When it encounters the first occurrence of a reference's small integer, it allocates the storage and records the new reference in the appropriate entry of the array, and proceeds to read the referent from the byte stream. When it encounters a subsequent occurrence of the reference's small integer, it just uses the reference obtained by indexing the table.

The default behavior of the pickle package isn't satisfactory for all types. Problems can arise if the concrete representation of an abstract type isn't an appropriate way to communicate the value between address spaces. To deal with this, the pickle package permits clients to specify custom procedures for pickling (and therefore for marshaling) particular data types. Typically the implementer of an abstract data type specifies such a custom procedure if the type's values aren't transferable by straightforward copying. We use this facility to marshal network objects, readers, and writers that are embedded in structures that would ordinarily be marshaled by value.

The narrowest surrogate rule places a serious constraint on the pickles design: since pickling and unpickling an object can make its type less specific, the unpickler must check that an unpickled object is legal at the position in which it occurs. It is possible for the unpickler to check this because the runtime provides the ability to enumerate the types of the fields of an object.

There are many subtleties in the design of the pickles package. The ability to register custom pickling procedures for selected types has a tricky interaction with subtyping. It is also tricky to define and efficiently compute type fingerprints for recursive and opaque types. But the details are beyond the scope of the present work.

2.3 Garbage collection

Our system includes network-wide, reference-counting garbage collection. For each network object, the runtime records the set of clients containing surrogates for the object (the *dirty set*). As long as this set is non-empty, the runtime retains a pointer to the object. The retained pointer protects the object from the owner's garbage collector, even if no local references to it remain. When a surrogate is created, the client runtime adds its address space to the dirty set for the concrete object by making a "dirty call" to the owner. When a surrogate is reclaimed, the client runtime deletes its address space from the dirty set by making a "clean call". When the dirty set becomes empty, the owner's runtime discards the retained pointer, allowing the owner's local garbage collector to reclaim the object if no local references remain. To trigger the clean call, the client runtime relies on the assumed ability to register cleanup hooks for surrogates with the local collector.

This scheme will not garbage-collect cycles that span address spaces. To avoid this storage leak, programmers are responsible for explicitly breaking such cycles.

If program `A` sends program `B` a reference to an object owned by a third program `C`, and `A` then drops its reference to the object, we must ensure that the dirty call from `B` precedes the clean call from `A`, to avoid the danger that the object at `C` will be prematurely collected. This is not a problem if the object is sent as an argument to a remote method call, since in this case the calling thread retains a reference to the object on its stack while it blocks waiting for the return message, which cannot precede the unmarshaling of the argument. But if the object is sent as a result rather than an argument, the danger is real. Our solution is to require an acknowledgement to any result message that contains a network object: the procedure that executes the call in the owner blocks waiting for the acknowledgement, with the reference to the object on its stack. The stack reference protects the object from the garbage collector. This solution increases the message count for method calls that return network objects, but it doesn't greatly increase the latency of such calls, since the thread waiting for the acknowledgement is not on the critical path.

By maintaining the set of clients containing surrogates rather than a simple count, we are able to remove clients from the dirty set when they exit or crash. The mechanism for detecting that clients have crashed is transport-specific, but for all reasonable transports there is some danger that a network partition that prevents communication between the owner and client will be misinterpreted as a client crash. In this case, the owner's object might be garbage collected prematurely. Because communication is unreliable, the risk of premature collection is inherent in any strategy that avoids storage leaks in long-running servers. Since we never reuse object IDs, we can detect premature collection if it occurs.

Dirty calls are synchronous with surrogate creation, but clean calls are performed in the background and can be batched. If a clean call fails, it will be attempted again. If a dirty call fails, the client schedules the surrogate to be cleaned (since the dirty call might have added the client to the dirty set before failing) and raises the exception `NetObj.Error`. Clean and dirty calls carry sequence numbers that increase monotonically with respect to any given client: the owner ignores any clean or dirty call that is out of sequence. This requires the owner to store a sequence number for each entry in the dirty set, as well as a sequence number for each client for which a call has failed. The sequence numbers for clients that have successfully removed themselves from the dirty set can be discarded.

A companion paper[3] presents the details of the collection algorithm and a proof of its correctness.

2.4 Transports

There are many protocols for communicating between address spaces (for example, shared memory, TCP, and UDP), and many irksome differences between them. We insulate the main part of the network object runtime from these differences via the abstract type `Transport.T`.

A `Transport.T` object generates and manages connections between address spaces. Different subtypes are implemented using different communication mechanisms. For example, a `TCPTransport.T` is a subtype that uses TCP.

Each subtype is required to provide a way of naming address spaces. A transport-specific name for an address space is called an *endpoint*. Endpoints are not expected to be human-sensible. Naming conventions ensure that an endpoint generated by one transport subtype will be meaningful only to other instances of the same subtype. (Some use the term "endpoint" in a weaker sense, meaning little more than a port number. For us, different instances of a program are identified by different endpoints.)

The `fromEndpoint` method of a `Transport.T` enables creation of connections to recognized endpoints. If `tr` is a `Transport.T` and `ep` is an endpoint recognized by `tr`, then `tr.fromEndpoint(ep)` returns a `Location` (described in the next paragraph) that generates connections to the address space named by `ep`. If `tr` doesn't recognize `ep`, then such an invocation returns `NIL`.

A `Location` is an object whose `new` method generates connections to a particular address space. When a client has finished using a connection, it should pass the connection to the `free` method of the location that generated it. This allows transports to manage the allocation and deallocation of connections so as to amortize the overhead of connection establishment and maintenance.

The system uses the type `StubLib.Conn` to represent connections. It is perfectly possible to implement a class of connection that communicates with datagrams according to a protocol that makes idle connections essentially free[4]. That is, in spite of its name, implementations of the type `StubLib.Conn` need not be connection-oriented in the standard sense of the term.

A connection `c` contains a reader `c.rd` and a writer `c.wr`. Connections come in pairs; if `c` and `d` are paired, whatever is written to `c.wr` can be read from `d.rd`, and vice-versa. Ordinarily `c` and `d` will be in different address spaces. Values are marshaled into a connection's writer and unmarshaled from a connection's reader. Since readers and writers are buffered, the marshaling code can treat them either as streams of bytes (most convenient) or as streams of datagrams (most efficient).

One of the two connections in a pair is the *client* side and the other is the *server* side. Transports are required to provide a thread that listens to the server side of a connection and calls into the network object runtime when a message arrives indicating the beginning of a remote call. This is called *dispatching*, and is described further below.

A connection is required to provide a way of generating a "back connection": the location `c.loc` must generate connections to the address space at the other side of `c`. If `c` is a server-side connection, the connections generated by `c.loc` have the opposite direction as `c`; if `c` is a client-side connection, they have the same direction as `c`.

A transport is responsible for monitoring the liveness of address spaces for which it has locations or connections. This is discussed in more detail later when we specify the interface to the transport system.

2.5 Basic representations

We will now describe the wire representation of network objects, the client and server stubs involved in remote invocation, and the algorithms we use to marshal and unmarshal network objects. In these descriptions we will use Modula-like pseudocode.

The wire representation for a network object is a pair `(sp, i)` where `sp` is a `SpaceID` (a number that identifies the owner of the object) and `i` is an `ObjID` (a number that distinguishes different objects with the same owner):

```
TYPE WireRep = RECORD sp: SpaceID; i: ObjID END;
```

Each address space maintains an *object table* `objtbl` that contains all its surrogates and all its network objects for which any other space holds a surrogate:

```
VAR objtbl: WireRep -> NetObj.T;
```

We use the notation `A -> B` to name the type of a table with domain type `A` and element type `B`. We will use array notation for accessing the table, even though it is implemented as a hash table. We write `domain(tbl)` to denote the set of elements of the domain of `tbl`.

We now specify the representation of the opaque type `NetObj.T`. In Modula-3, the `REVEAL` statement permits such a specification to be visible within a bounded scope.

```
REVEAL
```

```
NetObj.T = OBJECT
  srgt, inTbl: BOOLEAN;
  wrep: WireRep;
  loc: Location;
  disp: Dispatcher
END;

TYPE Dispatcher = PROC(c: StubLib.Conn; obj: NetObj.T);
```

The field `obj.srgt` indicates whether `obj` is a surrogate. The field `obj.inTbl` indicates whether `obj` is present in `objTbl`, and this is guaranteed to be the case if `obj` is a surrogate or if another address space holds a surrogate for it. If `obj.inTbl` is `TRUE`, then `obj.wrep` is the wire representation of the object.

If `obj` is a surrogate is then `obj.loc` is a `Location` that generates connections to the owner's address space at `obj.wrep.sp`, and `obj.disp` is unused. Otherwise, if `obj.inTbl` is `TRUE`, then `obj.disp` is the dispatcher procedure for the object, and `obj.loc` is unused. The call `obj.disp(c, obj)` unmarshals a method number and arguments from `c`, calls the appropriate method of `obj`, and marshals the result to `c`.

2.6 Remote invocation

To illustrate the steps in a remote method invocation we continue with our example of a simple file service. In that example, we defined the type `FS.Server` with a single method `open`. The corresponding stub-generated surrogate type declaration looks like this:

```
TYPE
  SrgSvr = FS.Server OBJECT
    OVERRIDES
      open := SrgOpen
    END;

PROCEDURE SrgOpen(obj: SrgSvr; n: TEXT): FS.File =
  VAR
    c := obj.loc.new();
    res: FS.File;
  BEGIN
    OutNetObj(c, obj);
    OutInteger(c, 0);
    OutText(c, n);
    <flush buffers to network>;
    res := InNetObj(c);
    obj.loc.free(c);
    RETURN res
  END SrgOpen;
```

The procedures `OutNetObj` and `InNetObj` are described in the next subsection. Procedures for marshaling basic types (like `OutInteger`) are in the `StubLib` interface which we specify later in the report. (Actually, `StubLib.OutRef` subsumes both `OutNetObj` and `OutText`, but we ignore that here.)

The method being invoked is identified on the wire by its index; the `open` method has index zero. The code presented would crash with a narrow fault if the network object returned by

`InNetObj` were not of type `FS.File`. For example, this would happen if appropriate stubs had not been linked into the client or owner. The actual system would raise an exception instead of crashing.

On the server side, the thread forked by the transport to service a connection `c` calls into the network object runtime when it detects an incoming RPC call. The procedure it calls executes code something like this:

```
VAR obj := InNetObj(c); BEGIN obj.disp(c, obj) END;
```

The dispatcher procedures are typically written by the stub generator. The dispatcher for the type `FS.Server` would look something like this:

```
PROCEDURE SvrDisp(c: StubLib.Conn; obj: FS.Server) =
  VAR methID := InInteger(c); BEGIN
    IF methID = 0 THEN
      VAR
        n := InText(c);
        res := obj.open(n);
      BEGIN
        OutNetObj(c, res);
        <flush buffers to network>
      END
    ELSE
      <error, non-existent method>
    END
  END SvrDisp;
```

The stubs have a narrow interface to the rest of the system: they call the `new` and `free` methods of `Location` objects to obtain and release connections, and they register their surrogate types and dispatcher procedures where the runtime can find them, in the global table `stubs`:

```
VAR stubs: Typecode -> StubRec;

TYPE StubRec = RECORD srgType: TypeCode; disp: Dispatcher END;
```

An address space has stubs for `tc` if and only if `tc` is in the domain of `stubs`. If `tc` is in the domain of `stubs`, then `stubs[tc].srgType` is the typecode for the surrogate type for `tc`, and `stubs[tc].disp` is the owner dispatcher procedure for handling calls to objects of type `tc`.

A stub module that declares a surrogate type `srgTC` and dispatcher `disp` for a network object type `tc` also sets `stubs[tc] := (srgTC, disp)`. The network object runtime automatically registers a surrogate type and null dispatcher for the type `NetObj.T`.

In the actual system the `stubs` table is indexed by stub protocol version as well as type code, to make it easy for a program to support multiple protocol versions. The actual system also includes code for relaying exceptions raised in the owner to the client, and for relaying thread alerts from the client to the owner.

2.7 Marshaling network objects

The call `OutNetObj(c, obj)` writes the wire representation of `obj` to the connection `c`:

```
PROCEDURE OutNetObj(c: StubLib.Conn; obj: NetObj.T) =
  BEGIN
    IF obj = NIL THEN
      OutWireRep(c, (-1,-1));
    ELSE
      IF NOT obj.inTbl THEN
        VAR i := NewObjID(); BEGIN
          obj.wrep := (SelfID(), i);
          objtbl[obj.wrep] := obj;
          obj.inTbl := TRUE;
          obj.srgt := FALSE;
          obj.disp := GetDisp(TYPECODE(obj))
        END
      END;
      OutWireRep(c, obj.wrep)
    END
  END OutNetObj;

PROCEDURE GetDisp(tc: INTEGER): Dispatcher =
  BEGIN
    WHILE NOT tc IN domain(stubs) DO tc := Supertype(tc) END;
    RETURN stubs[tc].disp
  END GetDisp;
```

In the above we assume that `NewObjID()` returns an unused object ID, that `SelfID()` returns the `SpaceID` of the caller, and that `Supertype(tc)` returns the code for the supertype of the type whose code is `tc`.

The corresponding call `InNetObj(c)` reads a wire representation from the connection `c` and returns the corresponding network object reference:

```
PROCEDURE InNetObj(c: StubLib.Conn): NetObj.T =
  VAR wrep := InWireRep(c); BEGIN
    IF wrep.i = -1 THEN
      RETURN NIL
    ELSIF wrep IN domain(objtbl) THEN
      RETURN objtbl[wrep]
    ELSE
      RETURN NewSrgt(wrep, c)
    END
  END InNetObj;
```

The call `NewSrgt(wrep, c)` creates a surrogate for the network object whose wire representation is `wrep`, assuming that `c` is a connection to an address space that knows `wrep.sp`. (We say that an address space `sp1` *knows* an address space `sp2` if `sp1=sp2` or if `sp1` contains some surrogate owned by `sp2`.)

`NewSrgt` locates the owner, determines the typecode of the surrogate, and enters it in the object table:

```
PROCEDURE NewSrgt(wrep: WireRep; c: StubLib.Conn): NetObj.T =
  VAR
    loc := FindSpace(wrep.sp, conn);
    tc := ChooseTC(loc, wrep.i);
```

```
    res := Allocate(tc);
  BEGIN
    res.wrep := wrep;
    res.srgt := TRUE;
    res.inTbl := TRUE;
    objtbl[wrep] := res;
    RETURN res
  END NewSrgt;
```

The call `FindSpace(sp, c)` returns a `Location` that generates connections to `sp`, or raises `NetObj.Error` if this is impossible. It requires that `c` be a connection to an address space that knows about `sp`. The call `ChooseTC(loc, i)` implements the narrowest surrogate rule. It returns the local code for the local surrogate type for the object whose ID is `i` and whose owner is the address space to which `loc` generates connections. The call `Allocate(tc)` allocates an object with type code `tc`.

To implement `FindSpace` without resorting to broadcast, each address space maintains information about its own transports and the endpoints of the address spaces it knows about. This information is maintained in the variables `tr` and `names`:

```
VAR tr: SEQ[Transport.T];
VAR names: SpaceID -> SEQ[Endpoint];
```

The sequence `tr` lists the transports available in this space, in decreasing order of desirability. Typically, it is initialized by the network object runtime and is constant thereafter. For any space `sp`, the sequence `names[sp]` contains the endpoints for `sp` recognized by `sp`'s transports. We write `SEQ[T]` to denote the type of sequences of elements of type `T`.

The fast path through `FindSpace` finds an entry for `sp` in `names`; this entry is the list of names for `sp` recognized by `sp`'s transports. These names are presented to the transports `tr` available in this space; if one is recognized, a common transport has been found; if none is recognized, there is no common transport.

The first time an address space receives a reference to an object owned by `sp`, there will be no entry for `sp` in the space's name table. In this case, `FindSpace` obtains the name sequence for `sp` by making an RPC call to the address space from which it received the reference into `sp`. This is our first example of an RPC call that is nested inside an unmarshaling routine; we will use the notation `RPC(loc, P(args))` to indicate an RPC call to `P(args)` directed at the address space identified by the location `loc`. Here is the implementation of `FindSpace`:

```
PROCEDURE FindSpace(sp: SpaceID; c: StubLib.Conn): Location =
  BEGIN
    IF NOT sp IN domain(names) THEN
      names[sp] := RPC(c.loc, GetNames(sp));
    END;
    VAR nm := names[sp]; BEGIN
      FOR i := 0 TO LAST(tr) DO
        FOR j := 0 TO LAST(nm) DO
          VAR loc := tr[i].fromEndpoint(nm[j]); BEGIN
            IF loc # NIL THEN RETURN loc END
          END
        END
      END;
```

```
      RAISE NetObj.Error
    END
  END FindSpace;

PROCEDURE GetNames(sp) = BEGIN RETURN names[sp] END GetNames;
```

Placing the `i` loop outside the `j` loop gives priority to the client's transport preference over the owner's transport preference. The choice is arbitrary: usually the only point of transport preference is to obtain a shared memory transport if one is available, and this will happen whichever loop is outside.

The only remaining procedure is `ChooseTC`, which must implement the narrowest surrogate rule. According to this rule, the surrogate type depends on which stubs have been registered in the client and in the owner: it must determine the narrowest supertype for which both client and owner have a registered stub. This requires a call to the owner at surrogate creation time, which we combine with the call required by the garbage collector: the call `Dirty(i, sp)` adds `sp` to the dirty set for object number `i` and returns the supertypes of the object's type for which stubs are registered in the owner.

```
PROCEDURE Dirty(i: ObjID; sp: SpaceID): SEQ[Fingerprint] =
  VAR
    tc := TYPE(objtbl[(SelfID(), i)]);
    res: SEQ[Fingerprint] := <empty sequence>;
  BEGIN
    <add sp to object i's dirty set>;
    WHILE NOT tc IN domain(stubs) DO tc := Supertype(tc) END
    LOOP
      res.addhi(TCToFP(tc));
      IF tc = TYPECODE(NetObj.T) THEN EXIT END;
      tc := Supertype(tc)
    END;
    RETURN res
  END Dirty;

PROCEDURE ChooseTC(loc: Location; i: ObjID): INTEGER =
  VAR fp: SEQ[Fingerprint]; BEGIN
    fp := RPC(c.loc, Dirty(i, SelfID()));
    FOR j := 0 TO LAST(fp) DO
      IF FPToTC(fp[j]) IN domain(stubs) THEN
        RETURN stubs(FPToTC(fp[j])).srgType
      END
    END
  END ChooseTC;
```

The loops in `Dirty` are guaranteed to terminate, because stubs are automatically registered for `NetObj.T`. In `ChooseTC` we assume that `TCToFP` and `FPToTC` convert between equivalent typecodes and fingerprints (if there is no local typecode for `fp`, we assume that `FPToTC(fp)` returns some illegal typecode never present in `stubs`). We also assume that `s.addhi(x)` extends the sequence `s` with the new element `x`.

This concludes our description of the algorithms for marshaling network objects. We have omitted a number of details. For example, to avoid cluttering up the program, we have ignored synchronization; the real program must protect the various global tables with locks. Some care is required to avoid deadlock; for example, it is not attractive to hold a lock all the way through a call to `NewSrgt`. Instead, we make an entry in the surrogate table at the beginning of the procedure, recording that a surrogate is under construction, and do not reacquire the table lock until the end of the sequence, when the surrogate has been fully constructed. A thread that encounters a surrogate under construction simply waits for it to be constructed.

2.8 Marshaling streams

An important feature of our treatment of streams is that data is not communicated via RPC, but by the underlying transport-specific communication. This facility is analogous to the remote pipes of DCE RPC, but with a critical difference: the streams we pass are not limited in scope to the duration of the RPC call. When we marshal a stream from process A to process B, process B acquires a surrogate stream attached to the same data as the original stream. In process B the surrogate stream can be used at will, long after the call that passed it is finished. In contrast, in a scheme such as the pipes provided in DCE, the data in the pipe must be communicated in its entirety at the time of the RPC call (and at a particular point in the call too). Our facility is also analogous to the remote pipes of Gifford and Glasser[10], but is simpler and more transparent.

Readers and writers are marshaled very similarly; to be definite, consider a reader `rd`. The sending process has a concrete reader `rd` in hand. The marshaling code must create a surrogate reader `rdsrg` in the receiving process, such that `rdsrg` delivers the contents of `rd`. The general strategy is to allocate a connection between the sender and receiver, allocate `rdsrg` in the receiver so that it reads from the connection, and fork a thread in the sender that reads buffers from `rd` and sends them over the connection. (The thread could be avoided by doing an RPC to fill the buffer of `rdsrg` whenever it is empty, but this would increase the per-buffer overhead of the cross-address space stream.) For the detailed strategy we explored two designs.

In the first design, the sender uses the connection `c`, over which `rd` is to be marshaled, to create a new connection `nc := c.loc.new()`. The sender then chooses a unique ID, sends the ID over `nc`, sends the ID over `c` as the wire representation of `rd`, and forks a thread that copies data from `rd` into `nc`. In the receiving process, two threads are involved. The thread servicing the connection `nc` reads the ID (distinguishing it from an incoming call message) and places the connection in a table with the ID as key. The thread unmarshaling the reader looks up the connection in the table and allocates the surrogate reader `rdsrg` using that connection. This seems simple, but the details became rather complicated, for example because of the difficulty of freeing connections in the table when calls fail at inopportune times.

The second design employs a network object called a `Voucher` with a method `claim` that returns a reader. Vouchers have nonstandard surrogates and dispatchers registered for them, but are otherwise ordinary network objects.

To marshal `rd`, the sending process allocates a voucher `v` with a data field `v.rd` of type reader, sets `v.rd := rd`, and calls `OutNetObj(v)`. When the receiving process unmarshals a network object and finds it is a surrogate reader voucher `vs`, it calls `vs.claim()` and returns the resulting reader.

The `claim` method of a surrogate voucher `vs` invokes `vs.loc.new()` to obtain a new connection `nc`. It then marshals `vs` to `nc` (just like an ordinary surrogate method call). But then, instead of sending arguments and waiting for a result, it allocates and returns the surrogate reader `rdsrg`, giving it the connection `nc` as a source of data.

On the server side, the voucher dispatcher is called by a transport-supplied thread, just as for an ordinary incoming call. The arguments to the dispatcher are the server side of the connection `nc` and the voucher `vs` containing the original reader `vs.rd`. The dispatcher procedure plays the role of the forked thread in the first design: it reads buffers from `vs.rd` and sends them over `nc`.

The second design relies on the transport to provide the required connection and thread, and relies on the ordinary network object marshaling machinery to connect the surrogate voucher with the original reader. This makes the protocol simple, but it costs three messages (a round trip for the dirty call for the voucher; then another message to launch the voucher dispatcher). It would be easy enough to avoid the all-but-useless dirty call by dedicating a bit in the wire representation to identify vouchers, but perhaps not so easy to stomach the change. On the other hand, the first design uses only one message (to communicate the `ID` from the sender to the receiver), and this message could perhaps be piggybacked with the first buffer of data.

We chose the second design, because: (1) it is trivial to implement; (2) given that cross-address space streams are intended for bulk data transfer, it is not clear how important the extra messages are; and (3) if experience leads us to get rid of the extra messages, it is not obvious whether to choose the first design or to optimize the second.

2.9 Bootstrapping

The mechanisms described so far produce surrogate network objects only as a result of method calls on other surrogate network objects. We have as yet no way to forge an original surrogate. To do this we need the ingredients of a surrogate object: a `Location`, an object ID, and a surrogate type. To make it possible to forge the object ID and type, we adopt the following convention: every program into which network objects are linked owns a *special object* with an ID of zero, of a known type. The methods of the special object implement the operations required by the network object runtime (reporting in clean and dirty, `GetNames`, etc.). The special object also has `get` and `put` methods implementing a table of named network objects. At initialization time the network object runtime allocates a special object with ID 0 and places it in `objtbl`.

All that remains to forge an original surrogate is to obtain a `Location` valid for some other program into which network objects have been linked. Fundamentally, the only way to obtain a `Location` is to call some transport's `fromEndpoint` method—that is, the program forging the surrogate must know an address where something is listening. For this step the application has two choices. We provide a network object agent that listens at a well-known TCP port; thus a surrogate for the agent's special object can be forged given the IP name of the node on which it is running. If every node runs the agent from its start-up script, then no other well-known ports are needed: applications can export their objects by putting them in the table managed by the agent's special object, and their clients can get them from the same table. If the application writer prefers not to rely on the agent, he can choose his own transport and well-known port and configure his program to listen at that port and to forge surrogates for the special objects at that port.

The procedures `NetObj.Import` and `NetObj.Export`, which appeared in our file server example, implement object bootstrapping. These procedures simply forge a surrogate for the special object of some agent process, and then invoke the `get` or `put` method on that surrogate.

3 Public Interfaces

In this section and the next, we present the major system interfaces exactly as they appear in the network objects programmers library. This section describes the `NetObj`, `NetStream`, and `NetObjNotifier` interfaces, as well as the stub generator. These interfaces are sufficient for most network objects clients. The following section presents important internal interfaces that are not used by most clients.

The interfaces in this section depend on a few local-level facilities from the SRC Modula-3 runtime library[18, 13]. We summarize these dependencies here:

- An `Atom.T` is a unique representation for a set of equal texts (like a Lisp atomic symbol). Atoms are often used to parameterize exceptions.
- An `AtomList.T` is a linked list of atoms. Atom lists are used for propagating lists of nested exception parameters up the call stack.
- A `Rd.T` represents an abstract data source. The `Rd` interface provides operations for reading data and re-positioning the stream.
- A `Wr.T` represents an abstract data sink. The `Wr` interface provides operations for writing data and re-positioning the stream.
- `Thread.Alerted` is the exception to be raised by an alerted thread.
- The `WeakRef` interface allows clients to register garbage collection finalization procedures for objects in the traced heap. In other words, a client can obtain notification just prior to collection of heap storage.

3.1 NetObj interface

This is the primary public interface for using network objects. Before listing the interface, here are a few definitions.

A *program instance* is an activation of a program. The same program can have many instances running concurrently or consecutively. A program instance can be thought of as an address space, although the design does not preclude the implementation of a program instance by a suite of address spaces.

Recall that an agent is a program that provides a table that maps names to network objects. Any program can be an agent, but every machine has a particular default agent. Owners typically make network objects available to clients by inserting them into an agent's table, using the procedure `NetObj.Export`. Clients typically use `NetObj.Import` to retrieve network objects from the table.

```
INTERFACE NetObj;

IMPORT Atom, AtomList, Thread;
```

```
TYPE
  T <: ROOT;
  Address <: REFANY;
```

`NetObj.T` is the root type of all network objects. A `NetObj.Address` designates a program instance.

```
PROCEDURE Locate (host: TEXT): Address
    RAISES {Invalid, Error, Thread.Alerted};
```

Return an address for the default agent at the machine whose human-sensible name is `host`.

The naming convention used by `Locate` is system-dependent. For example, in an Internet environment, `Locate("decsrc.pa.dec.com")` returns the address of the default agent on the machine `decsrc` in the DEC Palo Alto Internet domain.

`Locate` raises `Invalid` if it determines that `host` is not a valid name. It raises `Error` if it is unable to interpret the name or determine its validity, typically because it is unable to contact the naming authority, or if there is no standard agent running on the specified host.

```
PROCEDURE Export(name: TEXT; obj: T; where: Address := NIL)
    RAISES {Error, Thread.Alerted};
```

Set `table[name] := obj` *where* `table` *is the table provided by the agent whose address is* `where`*, or by the default agent for the local machine if* `where=NIL`*. This can be used with* `obj=NIL` *to remove an entry from the table.*

```
PROCEDURE Import(name: TEXT; where: Address := NIL): T
    RAISES {Error, Thread.Alerted};
```

Return `table[name]` *where* `table` *is the table provided by the agent whose address is* `where`*, or by the default agent for the local machine if* `where=NIL`*.* `Import` *returns* `NIL` *if* `table` *contains no entry for* `name`*.*

```
EXCEPTION Error(AtomList.T), Invalid;

VAR (*CONST*)
  CommFailure, MissingObject, NoResources,
  NoTransport, UnsupportedDataRep, Alerted: Atom.T;

END NetObj.
```

The exception `NetObj.Error` indicates that a failure occured during a remote method invocation. Every remote method should therefore include this exception in its raises clause. If `NetObj.Error` is not raised, then the invocation completed successfully. If it is raised, it may or may not have completed successfully. It is possible that an orphaned remote invocation continued to execute at the owner, while the client raised `NetObj.Error`.

The first atom in the argument to `NetObj.Error` explains the reason for the failure; any subsequents atoms provide implementation-specific detail. The atom `CommFailure` indicates communication failure, which might be network failure or a crash on a remote machine. The atom `MissingObject` indicates that some network object, either the one whose method is invoked or an argument to that method, has been garbage-collected by its owner. (This indicates that the owner mistakenly determined that one of its clients was dead.) `NoResources` indicates that the call failed because of a lack of resources, for example Unix file descriptors. `NoTransport` indicates that an attempt to unmarshal an object failed because the client and owner shared no common transport protocol implementation and were therefore unable to communicate. `UnsupportedDataRep` indicates a mismatch between the network representation of data and the ability of a receiver to handle it, for example a 64-bit `INTEGER` with non-zero high-order bits is not meaningful as an `INTEGER` on a 32-bit machine. `Alerted` indicates that a client thread was alerted in the middle of a remote call and that an orphaned remote computation might still be in progress. (Threads alerted in remote calls might also raise `Thread.Alerted`; in which case it is guaranteed that no orphans remain.) If the first atom in the argument list does not appear in this interface, a network object runtime error is indicated.

3.2 NetStream interface

The `NetStream` interface describes the marshaling of readers and writers, and provides procedures that you will need to use if you plan to reuse a stream after marshaling it.

The network object runtime allows subtypes of `Rd.T` and `Wr.T` to be marshaled as parameters and as results of remote method invocation. To communicate a reader or writer from one program to another, a surrogate stream is created in the receiving program. We call the original reader or writer the concrete stream. Data is copied over the network between the concrete stream and the surrogate stream. Surrogate streams are free-standing entities, valid beyond the scope of the remote call that produced them. Data can be transmitted on a surrogate stream at close to the bandwidth supported by the underlying transport.

The initial position of the surrogate reader or writer equals the position of the corresponding concrete stream at the time it was marshaled. All surrogate readers and writers are unseekable. Data is transferred between surrogates and concrete streams in background. Therefore, undefined behaviour will result if you 1) perform local operations on the concrete stream while a surrogate for it exists, or 2) create two surrogates for the same stream by marshaling it twice. There is a mechanism, described below, for shutting down a surrogate stream so that the underlying stream can be remarshaled.

Calling `Wr.Flush` on a surrogate writer flushes all outstanding data to the concrete writer and flushes the concrete writer. Calling `Wr.Close` flushes and then closes both the surrogate and the concrete writer. Similarly, a call on `Rd.Close` on a surrogate closes both readers.

Clients who marshal streams retain responsibility for closing them. For example, `Rd.Close` on a surrogate can fail due to the network, leaving the owner responsible for closing the concrete reader. The `WeakRef` interface can be used to register a GC cleanup procedure for this purpose.

The `ReleaseWr` procedure is used to shut down a surrogate writer so that the underlying writer can be reused. It flushes any buffered data, closes the surrogate, and frees any network resources associated with the surrogate. It leaves the concrete writer in a state where it can be reused locally or remarshaled.

Similarly. the `ReleaseRd` procedure is used to shut down a surrogate reader so that the underlying reader can be reused. It closes the surrogate, frees any network resources associated with the surrogate, and leaves the concrete reader in a state where it can be reused locally or remarshaled. There is an important difference between releasing readers and writers:

`ReleaseRd` discards any data buffered in the surrogate or in transit.

```
INTERFACE NetStream;

IMPORT Rd, Wr, Thread;

PROCEDURE ReleaseRd(rd: Rd.T)
    RAISES {Rd.Failure, Thread.Alerted};
```

If `rd` is a surrogate reader, release all network resources associated with `rd`, discard all buffered data, close `rd`, but do not close the concrete reader for `rd`. This procedure is a no-op if `rd` is not a surrogate.

```
PROCEDURE ReleaseWr(wr: Wr.T)
    RAISES {Wr.Failure, Thread.Alerted};
```

If `wr` is a surrogate writer, flush `wr`, release all network resources associated with `wr`, close `wr`, but do not close the concrete writer for `wr`. This procedure is a no-op if `wr` is not a surrogate.

```
END NetStream.
```

3.3 NetObjNotifier interface

The `NetObjNotifier` interface allows the holder of a surrogate object to request notification of when the object's owner becomes inaccessible. This can be useful, for example, if it is necessary to remove surrogates from a table upon termination of the programs holding their corresponding concrete objects.

```
INTERFACE NetObjNotifier;

IMPORT NetObj;

TYPE
  OwnerState = {Dead, Failed};

  NotifierClosure = OBJECT METHODS
    notify(obj: NetObj.T; st: OwnerState);
  END;

PROCEDURE AddNotifier(obj: NetObj.T; cl: NotifierClosure);
```

Arrange that a call to `cl.notify` will be scheduled when `obj` becomes inaccessible. If `obj` is not a surrogate object then `AddNotifier` has no effect. If `obj` is already inaccessible at the time `AddNotifier` is called, then a call to `cl.notify` is scheduled immediately.

```
END NetObjNotifier.
```

The `notify` method of a `NotifierClosure` object is invoked when the concrete object corresponding to the surrogate `obj` becomes inaccessible. The procedure `AddNotifier` must have been called to enable this notification. There may be more than one `NotifierClosure` for the same surrogate. At notification time, the `st` argument is `Dead` if and only if the object owner is known to be permanently inaccessible. Otherwise `st` is `Failed`. It is possible for `notify` to be called multiple times on the same object. Any invocations on `obj` are guaranteed to fail in a timely fashion subsequent to a closure notification with `st = Dead`.

In general, a surrogate object can still be collected if a notifier closure is registered for it. However, if the closure object contains a reference to the surrogate, then its registration might delay or prevent collection. Therefore this should be avoided.

Although this interface is organized to enable notification of owner death on a per object basis, in practice this is achieved by monitoring the state of the owner's address space. This means that death notification will be more or less simultaneous for all surrogates whose concrete objects have the same owner.

3.4 The stub generator

The stub generator is a program that generates stubs for Modula-3 network object types. There are restrictions on the subtypes of `NetObj.T` for which the stub generator can produce stubs; a network object type that obeys them is said to be *valid*. Here is a list of these restrictions:

1. A valid network object type must be pure, that is it cannot contain data fields, either in its declaration or in a revelation.
2. To generate stubs for a network object `I.T`, the stub generator must be able to determine a complete revelation for all opaque supertypes of `I.T` (including `I.T` itself, if it is opaque) up to `NetObj.T`.
3. A method argument may not be of type `PROCEDURE` or have a component that is of type `PROCEDURE`. (A network object with an appropriate method can always be sent instead of a procedure.)
4. A Modula-3 method declaration specifies the set of exceptions that the method can raise. It is possible to specify (via `RAISES ANY`) that any exception can be raised, but this is not allowed for a valid network object type.
5. The methods of the type and its supertypes must have distinct names.

Given a valid network object type, the stub generator lays down code that implements parameter marshaling and remote invocation for that type's methods. For both arguments and results, subtypes of `NetObj.T` are marshaled as network references, subtypes of `Rd.T` and `Wr.T` are marshaled as surrogate streams, and all other parameters are marshaled by copying. The copying is performed by the pickles package if the parameter is a reference.

`VALUE` and `READONLY` parameters are copied only once, from the caller to the owner of the object. `VAR` parameters are normally copied from caller to owner on the call, and from owner to caller when the call returns. The pragma `<*OUT*>` on a `VAR` parameter in a method declaration indicates that the parameter may be given an arbitrary legal value when the method is invoked. The stub generator may use this information to optimize method invocation by not copying the parameter's value from caller to owner. At present, the stub generator does not make this optimization.

Any change in marshaling protocol that would make stubs incompatible is implemented as a new version of the stub generator. Typically, the previous version will continue to be supported for some time after the release of a new one. Thus, multiple versions of the stub generator may sometimes exist at the same time.

Stubs for multiple versions may be linked into the same program. Method invocation between two programs is possible so long as the owner and the caller have at least one common version of the stubs for the network object in question. The network object runtime will use the most recent version of the protocol that is available in both programs. This allows gradual migration of applications from the old to the new protocol.

4 Internal Interfaces

In this section we present the main internal systems interfaces. The typical programmer using network objects has no need to read them, but we present them here in order to document the structure of the system. These are the interfaces you would use to write a new stub generator, hand-code stubs for some particular network object type, or add a new transport to the system.

4.1 StubLib interface

This interface contains procedures to be used by stub code for invoking remote object methods and servicing remote invocations. Each stub module provides type-dependent network support for marshaling and unmarshaling method calls for a specific subtype of `NetObj.T`. Usually, stubs are built automatically. For each `NetObj.T` subtype `T` intended to support remote method invocation there must be both a client and a server stub. The client stub defines a subtype of `T` in which every method is overridden by a procedure implementing remote method invocation. Such a surrogate object is constructed by the network object runtime whenever a reference to a non-local object is encountered. The server stub consists of a single procedure of type `Dispatcher` that is called to unmarshal and dispatch remote invocations. A surrogate type and null dispatcher for `NetObj.T` are defined and registered by the network object system itself.

```
INTERFACE StubLib;

IMPORT Atom, AtomList, NetObj, Rd, Wr, Thread;

TYPE Conn <: ROOT;
```

A remote object invocation can be viewed as an exchange of messages between client and server. The messages are exchanged via an object of type `Conn`, which is opaque in this interface. The `StubConn` interface reveals more of this type's structure to clients who wish to hand-code stubs for efficiency. A `Conn` is unmonitored: clients must not access it from two threads concurrently.

```
TYPE
  Byte8 = BITS 8 FOR [0..255];
  DataRep = RECORD
    private, intFmt, floatFmt, charSet: Byte8;
  END;

VAR (*CONST*) NativeRep: DataRep;
```

The type `DataRep` describes the format used to encode characters, integers, and floating point numbers in network data. Data is always marshaled in the sender's native format. `NativeRep` is a runtime constant that describes the native format of the current environment.

Stubs may optimize in-line unmarshaling by first checking that the incoming representation is the same as the native one for all data types relevant to the call. If it is not, then the generic data unmarshaling routines at the end of this interface should be used.

Automatic conversion between the data representations is performed wherever possible. If conversion is impossible, `NetObj.Error` is raised with `NetObj.UnsupportedDataRep` in the argument atom list.

Concrete values for the elements of `DataRep` are not defined here as it is sufficient to compare against `NativeRep` and invoke the marshaling procedures defined below if the encoding is non-native.

```
TYPE
  Int32 =  BITS 32 FOR [-16_7FFFFFFF-1..16_7FFFFFFF];
  StubProtocol = Int32;
CONST
  NullStubProtocol = -1;
  SystemStubProtocol = 0;
```

The type `StubProtocol` indicates the version of the stub compiler used to generate a particular stub. Multiple stubs for the same network object can coexist within the same program (for example, the outputs of different stub compilers). During surrogate creation, the network object runtime negotiates the stub protocol version with the object owner.

`NullStubProtocol` is a placeholder to indicate the absence of a stub protocol value. The value `SystemStubProtocol` indicates the fixed stub encoding used by the runtime to implement primitives that operate prior to any version negotiation.

```
VAR (*CONST*) UnmarshalFailure: Atom.T;
```

`UnmarshalFailure` should be used as an argument to `NetObj.Error` whenever stubs encounter a network datum that is incompatible with the target type. For example, the stub code might encounter a `CARDINAL` value greater than `LAST(CARDINAL)` or an unrecognized remote method specification.

```
TYPE Typecode = CARDINAL;
```

`Typecode` is the type of those values returned by the Modula-3 `TYPECODE` operator.

```
PROCEDURE Register(
    pureTC: Typecode; stubProt: StubProtocol;
    surrTC: Typecode; disp: Dispatcher);
```

Let `T` be the type whose typecode is `pureTC`, and let `srgT` be the type whose typecode is `surrTC`. Set the client surrogate type and dispatch procedure for `T` to be `srgT` and `disp`, respectively. The `stubProt` parameter indicates the stub compiler version that generated the stub being registered.

The following constraint applies to stub registration. If stubs are registered for types `A` and `B`, where `B` is a supertype of `A`, then the protocol versions registered for `B` must be a superset of the versions registered for `A`. If this rule is violated, attempts to invoke remote methods may raise `NetObj.Error`.

Note that a concrete object of type `A` will receive method invocations only for stub versions for which `A` is registered. This is true even if a supertype of `A` is registered with additional stub versions.

`Register` must be called before any object of type `T` is marshaled or unmarshaled.

Client stub procedures

Here is a simplified sketch of the procedure calls performed by a client to make a remote call to a method of `obj`:

```
VAR
  c := StartCall(obj, stubProt);
  resDataRep: DataRep;
BEGIN
  <marshal to "c" the number of this method>
  <marshal to "c" the method arguments>
  resDataRep := AwaitResult(conn);
  <unmarshal from "c" the method results>
  <results will be in wire format "resDataRep">
  EndCall(c, TRUE)
END;
```

For both arguments and results, the sender always marshals values in its native format; the receiver performs any conversions that may be needed. The procedure result typically begins with an integer specifying either a normal return or an exceptional return. If a protocol error occurs, the client should call `EndCall(c, FALSE)` instead of `EndCall(c, TRUE)`. This requires `TRY FINALLY` instead of the simple straight-line code above; a more complete example is presented in the next section.

Here are the specifications of the client protocol procedures:

```
PROCEDURE StartCall(obj: NetObj.T; stubProt: StubProtocol): Conn
    RAISES {NetObj.Error, Wr.Failure, Thread.Alerted};
```

Return a connection to the owner of `obj`, write to the connection a protocol request to perform a remote method call to `obj`, using the data representation `NativeRep`. The value `stubProt` is the stub protocol version under which the arguments and results will be encoded.

Upon return from `StartCall`, the client stub should marshal a specification of the method being invoked followed by any arguments.

```
PROCEDURE AwaitResult(c: Conn): DataRep
    RAISES {NetObj.Error, Rd.Failure, Wr.Failure,
            Thread.Alerted};
```

`AwaitResult` indicates the end of the arguments for the current method invocation, and blocks waiting for a reply message containing the result of the invocation. It returns the data representation used to encode the result message.

Upon return from `AwaitResult` the client stub should unmarshal any results.

```
PROCEDURE EndCall(c: Conn; reUse: BOOLEAN)
    RAISES {NetObj.Error, Rd.Failure, Wr.Failure,
            Thread.Alerted};
```

`EndCall` must be called at the end of processing a remote invocation, whether or not the invocation raised an exception. The argument `reUse` must be `FALSE` if the client has been unable, for any reason, to unmarshal either a normal or exceptional result. It is always safe to call `EndCall` with `reUse` set to `FALSE`, but performance will be improved if `reUse` is `TRUE` whenever possible.

`EndCall` determines, by examining `c`, whether the result message requires acknowledgement, that is, whether the result contained any network objects. If an acknowledgement is required, it is sent. `EndCall` then releases `c`. After `EndCall` returns, `c` should not be used.

Server dispatcher procedures

Next we consider the server-side stub, which consists of a registered dispatcher procedure.

```
TYPE Dispatcher = PROCEDURE(
  c: Conn; obj: NetObj.T; rep: DataRep; stubProt: StubProtocol)
  RAISES {NetObj.Error, Rd.Failure, Wr.Failure, Thread.Alerted};
```

A procedure of type `Dispatcher` is registered for each network object type `T` for which stubs exist. The dispatcher is called by the network object runtime when it receives a remote object invocation for an object of type `T`. The `rep` argument indicates the data representation used to encode the arguments of the invocation. The `stubProt` argument indicates the version of stub protocol used to encode the call arguments. The same protocol should be used to encode any results.

The dispatcher procedure is responsible for unmarshaling the method number and any arguments, invoking the concrete object's method, and marshaling any results.

Here is a simplified sketch of a typical dispatcher:

```
PROCEDURE Dispatch(c, obj, rep) =
  BEGIN
    <unmarshal from "c" the method number>
    <unmarshal from "c" the method arguments>
    <arguments will be in the wire format "rep">
    <call the appropriate method of "obj">
    StartResult(c);
    <marshal to "c" the method result or exception>
  END Dispatch;
```

Here is the specification of `StartResult`:

```
PROCEDURE StartResult(c: Conn)
    RAISES {Wr.Failure, Thread.Alerted};
```

`StartResult` must be called by the server stub to initiate return from a remote invocation before marshaling any results.

Upon return from `StartResult` the stub code should marshal any results or error indications.

Marshaling of reference types

The following procedures are made available for marshaling of subtypes of `REFANY`.

```
PROCEDURE OutRef(c: Conn; r: REFANY)
    RAISES {Wr.Failure, Thread.Alerted};
```

Marshal the data structure reachable from `r`. Certain datatypes are handled specially: subtypes of `NetObj.T` are marshaled as network references. Subtypes of `Rd.T` and `Wr.T` are marshaled as surrogate streams. The types `TEXT` and `REF ARRAY OF TEXT` are marshaled by copying via custom code for speed. All others are marshaled by copying as pickles. Subtypes of `NetObj.T`, `Rd.T`, and `Wr.T` which are embedded within other datatypes are also marshaled by reference.

```
PROCEDURE InRef(c: Conn; rep: DataRep; tc:=-1): REFANY
    RAISES {NetObj.Error, Rd.Failure, Thread.Alerted};
```

Unmarshal a marshaled subtype of `REFANY` as pickled by `OutRef`. If `tc` is non-negative, it is the typecode for the intended type of the reference. The exception `NetObj.Error(UnmarshalFailure)` is raised if the unpickled result is not a subtype of this type. If `tc` is negative, no type checking is performed.

`OutRef` and `InRef` use pickles and therefore are affected by any custom pickling procedures that have been registered. The network objects runtime itself registers procedures for pickling network objects and streams. Therefore, for any network objects or streams that are reachable from the reference `r` are pickled by reference as described elsewhere in this report.

Marshaling of generic data

The `StubLib` interface also provides a suite of procedures to facilitate the marshaling and unmarshaling of primitive data types. For the sake of brevity, we use the `INTEGER` datatype as an example. The actual interface provides routines to handle all other types that are primitive in Modula-3 such as `CARDINAL`, `REAL`, and `LONGREAL`.

```
PROCEDURE OutInteger(c: Conn; i: INTEGER)
    RAISES {Wr.Failure, Thread.Alerted};

PROCEDURE InInteger(c: Conn; rep: DataRep;
    min := FIRST(INTEGER); max := LAST(INTEGER)): INTEGER
    RAISES {NetObj.Error, Rd.Failure, Thread.Alerted};
```

Since all marshaling procedures output their parameters in the native representation of the sender, they can be trivially replaced by inline code that manipulates the writer buffer directly. All unmarshaling procedures decode the incoming wire representation as indicated by `rep` and return their results in native format. These procedures can be replaced by inline unmarshaling code whenever the relevant elements of `rep` match the corresponding elements of `NativeRep`.

Finally, the `StubLib` interface provides two procedures for raising `NetObj` exceptions conveniently:

```
PROCEDURE RaiseUnmarshalFailure() RAISES {NetObj.Error};
```

Raise `NetObj.Error` with UnmarshalFailure in the argument list.

```
PROCEDURE RaiseCommFailure(e: AtomList.T) RAISES {NetObj.Error};
```

Raise `NetObj.Error` with the result of prepending `NetObj.CommFailure` to `e`.

```
END StubLib.
```

4.2 An example stub

This subsection illustrates the use of the `StubLib` interface by presenting hand-generated stub code for a simple network object type, `Example.T`:

```
INTERFACE Example;

IMPORT NetObj, Thread;

EXCEPTION Invalid;

TYPE
  T = NetObj.T OBJECT METHODS
    get(key: TEXT) : TEXT
        RAISES {Invalid, NetObj.Error, Thread.Alerted};
  END;

END Example.
```

Notice that the object methods must raise `NetObj.Error` or else communications failures will be treated as checked runtime errors. Also notice that `Thread.Alerted` is present in the `RAISES` clause of all methods. This is not required, but is strongly advised. If an object method does not propagate the `Thread.Alerted` exception, then not only is it impossible to alert remote invocations, but the server implementation must guarantee that `Alerted` will never be raised. This guarantee must hold even though the network object runtime uses `Thread.Alert` to recover server threads when the client address space dies.

The following module defines and registers both client and server stubs for `Example.T`:

```
MODULE Example;

IMPORT NetObj, StubLib, Thread, Rd, Wr;

TYPE P = { Get };  R = { OK, Invalid };
```

The enumerated types `P` and `R` define values to be associated with the methods of `T` and with the various results (normal return or exception) of these methods.

```
TYPE StubT = T OBJECT OVERRIDES get := SurrogateGet; END;
```

The type `StubT` is the surrogate object type for `T`. It provides method overrides that perform remote invocation.

```
CONST StubVersion = StubLib.SystemStubProtocol;
```

This constant will be set by the stub generator to denote the stub generator version that created a given stub.

```
PROCEDURE SurrogateGet (t: StubT; key: TEXT) : TEXT
    RAISES {Invalid, NetObj.Error, Thread.Alerted} =
  VAR reuse := FALSE;
      rep: StubLib.DataRep;
      c: StubLib.Conn;
      res: TEXT;
  BEGIN
    TRY
      c := StubLib.StartCall(t, StubVersion);
      TRY
        StubLib.OutInt32(c, ORD(P.Get));
        StubLib.OutRef(c, key);
        rep := StubLib.AwaitResult(c);
        CASE StubLib.InInt32(c, rep) OF
        | ORD(R.OK) =>
            res := StubLib.InRef(c, rep, TYPECODE(TEXT));
            reuse := TRUE;
        | ORD(R.Invalid) =>
            reuse := TRUE;
            RAISE Invalid;
        ELSE
            StubLib.RaiseUnmarshalFailure();
        END;
      FINALLY
        StubLib.EndCall(c, reuse);
      END;
    EXCEPT
    | Rd.Failure(ec) => StubLib.RaiseCommFailure(ec);
    | Wr.Failure(ec) => StubLib.RaiseCommFailure(ec);
    END;
    RETURN res;
  END SurrogateGet;
```

`Invoke` is the server stub dispatcher for `T`. It is called when the network object runtime receives a method invocation for an object of type `T`.

```
PROCEDURE Invoke(
    c: StubLib.Conn; obj: NetObj.T; rep: StubLib.DataRep;
    <*UNUSED*> stubProt: StubLib.StubProtocol)
    RAISES {NetObj.Error, Rd.Failure, Wr.Failure,
            Thread.Alerted} =
  VAR t := NARROW(obj, T);
  BEGIN
    TRY
      CASE StubLib.InInt32(c, rep) OF
      | ORD(P.Get) => GetStub(c, t, rep);
      ELSE StubLib.RaiseUnmarshalFailure();
```

```
      END;
    EXCEPT
    | Invalid =>
        StubLib.StartResult(c);
        StubLib.OutInt32(c, ORD(R.Invalid));
    END;
  END Invoke;
```

There is one server side stub procedure for each method of `T`.

```
PROCEDURE GetStub (c: StubLib.Conn; t: T; rep: StubLib.DataRep)
    RAISES {Invalid, NetObj.Error, Rd.Failure,
             Wr.Failure, Thread.Alerted} =
  VAR key, res: TEXT;
  BEGIN
    key := StubLib.InRef(c, rep, TYPECODE(TEXT));
    res := t.get(key);
    StubLib.StartResult(c);
    StubLib.OutInt32(c, ORD(R.OK));
    StubLib.OutRef(c, res);
  END GetStub;
```

All stub code is registered with the network object runtime by the main body of the stub module. The protocol number is set to the stub protocol constant defined above.

```
BEGIN
  StubLib.Register(
    TYPECODE(T), StubVersion, TYPECODE(StubT), Invoke);
END Example.
```

4.3 StubConn interface

A `StubLib.Conn` represents a bidirectional connection used to invoke remote methods by the network objects runtime. Here we reveal that a connection `c` consists of a message reader `c.rd` and a message writer `c.wr`.

Connections come in matching pairs; the two elements of the pair are typically in different address spaces. If `c1` and `c2` are paired, the target of `c1.wr` is equal to the source of `c2.rd`, and vice versa. Thus the messages written to `c1.wr` can be read from `c2.rd`, and vice versa.

```
INTERFACE StubConn;

IMPORT MsgRd, MsgWr, StubLib;

REVEAL StubLib.Conn <: Public;

TYPE Public = OBJECT rd: MsgRd.T; wr: MsgWr.T END;

END StubConn.
```

The types `MsgWr.T` and `MsgRd.T` are subtypes of the standard Modula-3 stream types `Wr.T` and `Rd.T`; they are described in detail in the next subsection. Since they are subtypes, any of the standard stream operations can be used on them. For example, in a hand-coded stub you could replace the pair

```
StubLib.OutByte(c, byte)
b := StubLib.InByte(c)
```

with the pair

```
Wr.PutChar(c.wr, VAL(byte, CHAR))
b := ORD(Rd.GetChar(c.rd)).
```

The gain in speed from this change will be very modest. To make optimization worthwhile, you will want to make direct access to the buffers in the reader and writer. To do this, import the `RdClass` and `WrClass` interfaces[18]. Importing these interfaces will allow you to write stubs that operate directly on the reader and writer buffers.

If you use this optimization, you will have to be careful about locks. All readers and writers contain an internal lock used to serialize operations. It is a requirement of the `StubLib` interface that all parameters of type `Conn` be passed with both streams unlocked. It is a further requirement that no client thread operate on the streams while an activation of a `StubLib` procedure is in progress.

4.4 Message readers and writers

The byte streams of the readers and writers in a `StubLib.Conn` are divided into segments called *messages*. Messages are convenient for delineating call and return packets, and seem essential for sending both data and control information for surrogate streams.

We define the types `MsgRd.T` and `MsgWr.T` to present the abstraction of a stream of messages. A message is a sequence of bytes terminated by an end-of-message marker. The initial position is at the start of the first message. Messages can be of zero length.

If the end-of-message marker is encountered while reading from a `MsgRd.T`, it is represented by `EndOfFile` on the reader. The `nextMsg` method can be used to advance to the next message in the stream. This method waits for the next message and returns `TRUE` when it becomes available. A return value of `FALSE` indicates that there are (and will be) no further messages. The reader's current position is set to zero on return from `nextMsg`, and the reader no longer reports `EndOfFile` (unless of course the next message is zero length).

If `nextMsg` is invoked when the reader is not at `EndOfFile`, the remaining bytes in the current message are skipped.

As for all readers, calling `Rd.Close` on a `MsgRd.T` releases all associated resources.

Here is a listing of the interface:

```
INTERFACE MsgRd;

IMPORT Thread, Rd;

TYPE
  T = Rd.T OBJECT METHODS
    nextMsg(): BOOLEAN RAISES {Rd.Failure, Thread.Alerted};
  END;

END MsgRd.
```

As with a MsgRd.T, the nextMsg method of a MsgWr.T can be used to end the current message and position the writer at the start of the next message. The writer's current position is reset to zero on return from nextMsg.

Invoking Wr.Flush on a MsgWr.T flushes the current buffer to the abstract writer target, but does not end the current message.

As for all writers, calling Wr.Close on a MsgWr.T releases all associated resources. Close also flushes and terminates the current message. This means that a zero-length message is sent at close time if no data has been written into the current message (for example, directly after nextMsg or writer initialization).

Here is a listing of the interface:

```
INTERFACE MsgWr;

IMPORT Thread, Wr;

TYPE
  T = Wr.T OBJECT METHODS
    nextMsg() RAISES {Wr.Failure, Thread.Alerted};
  END;

END MsgWr.
```

There are two final clauses in the specification of message readers and message writers. First, their buffers must be word-aligned in memory. More precisely, if byte i in the data stream is stored in the buffer at memory address j, then i and j must be equal modulo the machine word size. This requirement allows optimized stubs to read and write scalar word values from the buffer efficiently. Second, their buffers must not be too small. More precisely, when the nextMsg method of a writer returns, there must be at least 24 bytes of free space in the writer buffer, and when the nextMsg method of a reader returns, there must be at least 24 bytes of message data in the reader buffer. This requirement allows the runtime to efficiently read and write the headers required by the network object protocol.

4.5 Transport interface

The Transport interface separates the main part of the network object runtime system from the parts that deal with low-level communication. It is the interface that must be implemented to extend the system to use new communication protocols. The interface is reasonably narrow:

```
INTERFACE Transport;

IMPORT NetObj, NetObjNotifier, StubLib, StubConn, Thread;

TYPE
  T <: Public;
  Endpoint = TEXT;
  Public = OBJECT METHODS
    fromEndpoint(e: Endpoint): Location;
    toEndpoint(): Endpoint;
    serviceCall(t: StubLib.Conn): (*reUse*) BOOLEAN
      RAISES {Thread.Alerted};
  END;
```

```
  Location <: LocationP;
  LocationP = OBJECT METHODS
    new(): StubLib.Conn RAISES {NetObj.Error, Thread.Alerted};
    free(c: StubLib.Conn; reUse: BOOLEAN);
    dead(st: NetObjNotifier.OwnerState);
  END;

  Conn = StubConn.Public BRANDED OBJECT
    loc: Location
  END;

REVEAL
  NetObj.Address = BRANDED REF ARRAY OF Endpoint;
  StubLib.Conn <: Conn;

END Transport.
```

The main ideas in the interface were described earlier. To summarize these briefly:

- A `Transport.T` is an object that manages connections of some particular class (*e.g.*, TCP).
- A `Transport.Location` is an object that creates connections of some particular class to some particular address space.
- A `Transport.Endpoint` is a transport-specific name for an address space (*e.g.*, an IP address plus a port number plus a non-reusable process ID).
- The `fromEndpoint` method of a transport converts an endpoint into a location, or into `NIL` if the endpoint and transport are of different classes.

Here are specifications for the methods of a `Transport.T`:

- The `toEndpoint` method returns an endpoint for the address space itself. The resulting endpoint should be recognized by the `fromEndpoint` method of transports of the same class anywhere in the network. That is, if program instance `P` calls `tr.toEndpoint()`, producing an endpoint `ep`, then the call `tr1.fromEndpoint(ep)` executed in any program instance either returns `NIL` (if `tr` and `tr1` are of different classes) or returns a location that generates connections to `P`.
- Transports are required to provide the threads that listen to the server sides of connections. When a message arrives on the connection indicating the beginning of a remote call, the threads are required to call the `serviceCall` method of their transport. The default value of this method locates and calls the dispatcher procedure. Ordinarily a transport implementation will not need to override the `serviceCall` method. If `conn` is the argument to `serviceCall`, then at entry `conn.rd` is positioned at the start of the incoming message. The `serviceCall` method processes the incoming remote invocation and sends the result on `conn.wr`. If it returns `TRUE`, then the remote invocation was processed without error and the transport can cache the connection. If it returns `FALSE`, a protocol error occurred during the call, and the transport implementation should destroy the connection.

And here are the specifications for the methods of a `Transport.Location`:

- The `new` method of a location returns a connection to the address space for which it is a location. The call `loc.new()` returns a connection whose server side is that address space and whose client side is the program instance making the call. The caller must pass the resulting connection to `loc.free` when it is finished with it.

- The call `loc.free(c, reUse)` frees the connection `c`, which must have been generated by `loc.new()`. If `reUse` is `TRUE`, the client asserts that the connection is in a suitable state for executing another remote method call. In particular, `c.wr` must be positioned at the beginning of a message.

- A transport is responsible for monitoring the liveness of program instances for which it has locations or connections. The method of monitoring depends on the transport. For example, the transport might periodically ping the other program instances. A program is considered dead if it exits, crashes, or if the underlying communication network cannot reach it for an appreciable amount of time. Suppose that `loc` is a location that generates connections to some program instance `P`. If `P` dies, the transport that provided `loc` is responsible for calling the method `loc.dead(st)`. (The network object runtime implements this method; the transport should not override it.) The argument `st` indicates whether the transport has detected a permanent failure, or one that is potentially transient. In addition to calling `loc.dead`, the transport is responsible for alerting all threads it has spawned to handle method invocations on behalf of `P`.

A transport is expected to manage the connections it creates. If creating connections is expensive, then the transport's locations should cache them. If maintaining idle connections is expensive, then the transport's locations should free them. Often connections are time-consuming to create, but then tie up scarce kernel resources when idle. Therefore transports typically cache idle connections for a limited amount of time.

The `Transport` interface reveals the representation of `NetObj.Address`: an address is simply an array of endpoints for the program instance designated by the address. The endpoints are generally of different transport classes; they provide alternative ways of communicating with the program instance. The modules of the network object runtime that require this revelation are exactly the modules that import the transport interface, so this is a convenient place to put it.

The `Transport` interface also reveals more information about the type `StubLib.Conn`. If `t` is a `StubLib.Conn`, then `t.loc` is a `Location` that generates connections to the program instance at the other end of `t`. The connections generated by `t.loc` connect the same pair of program instances that `t` connects, but if `t` is a handle on the server side of the connection, then the connections generated by `t.loc` will reverse the direction of `t`: their client side will be `t`'s server side, and vice versa (so-called back connections). On the other hand, if `t` is a handle on the client side of the connection, then the connections generated by `t.loc` will be in the same direction as `t`. A transport must ensure that the `loc` field is defined in all connections returned by any of its locations.

5 Performance

Our system was designed and implemented in a year by the four authors. The network object runtime is 4000 lines, the stub generator 3000 lines, the TCP transport 1500 lines, the pickle package 750 lines, and the network object agent 100 lines. All the code is in Modula-3.

We haven't attempted extensive performance optimization, but we have measured the times for some basic operations. The numbers given in Table 1 were taken using Digital workstations equipped with DECchip 21064 processors (at 175 MHz) running OSF/1 and communicating over a 100 megabit/sec AN1 network[22]. The numbers include the cost of Modula-3 runtime checks.

Table 1: Sample remote invocation timings

Call parameters	*Elapsed time/call*
Null call	960 usec
Ten integer arguments	1010 usec
`REF CHAR` argument	1280 usec
Linked list argument	5200 usec
Network object argument (s)	1030 usec
Network object argument (c)	1050 usec
Network object argument (c, d)	2560 usec
Network object result (s)	1180 usec
Network object result (c)	1190 usec
Network object result (c, d)	2680 usec

(c) concrete object marshaled

(s) surrogate object marshaled

(d) dirty call required

On our test configuration, it takes 660 microseconds for a C program to echo a TCP packet from user space to user space. A null network object method invocation takes an additional 300 microseconds. The difference is primarily due to the cost of two Modula-3 user space context switches (64 microseconds), the cost of marshaling and unmarshaling the object whose null method is being invoked, and the cost of the wire protocol used to frame invocation and result packets.

The ten integer argument test shows that the incremental cost of an integer argument is about 5 microseconds. The `REF CHAR` test measures the cost of marshaling a small data structure by pickling. This minimal use of the pickle machinery adds an additional 220 microseconds to the null call. The linked list test measures the cost of marshaling a complex data structure, in this case a doubly-linked list with 25 elements. The additional cost per element is roughly 80 microseconds.

The next six tests show the total cost of various calls involving a single network object argument or result (in addition to the object whose method is being invoked). An "(s)" indicates that the argument or result is marshaled as a surrogate and unmarshaled as a concrete object. A "(c)" indicates that the argument or result is marshaled as a concrete object and unmarshaled as a surrogate. A “(d)” indicates that a dirty call is required.

The incremental cost of a surrogate network object argument that does not lead to a dirty call is roughly 70 microseconds. A concrete network object argument is somewhat more expensive. If a dirty call is required, there is an additional cost of about 1500 microseconds.

Network object results are more expensive than arguments, because of the acknowledgement that must be sent when the result message contains a network object. In the tests that do not involve dirty calls, this cost shows up as a difference of approximately 150 microseconds, but in the tests that do involve dirty calls the cost seems to be lost in the noise.

We also measured the performance of marshaled readers and writers. Since there is only a minimal layer of protocol between marshaled data streams and the underlying network transport, there is little difference in bandwidth. In the test configuration described above, our implementation delivers over 95 percent of the full network bandwidth (100 MBits/sec). We attribute our failure to achieve full network bandwidth to the cost of user-space thread emulation, Unix non-blocking I/O, and TCP protocol overhead.

The purpose of our project was to find an attractive design, not to optimize performance, and our numbers reflect this. Nevertheless, the performance of our system is adequate for many purposes. It is competitive with the performance of commercially available RPC systems[20], and we believe that our design does not preclude the sort of performance optimizations reported in the literature[23, 27]. Furthermore, our use of buffered streams for marshaling permits careful hand-tuning of stubs while still offering the flexibility of a general purpose stream abstraction.

6 Experience

Our network objects system has been working for almost two years. Several projects have built on the system, including the Siphon distributed software repository[21], the Argo teleconferencing system[9], and the Obliq distributed scripting language[6]. We report on experience gained from these projects here.

6.1 Siphon

The Siphon system consists of two major components. The *packagetool* allows software packages to be checked in and out from a repository implemented as a directory in a distributed file system. The repository is replicated for availability. When a new version of a package is checked in, it is immediately visible to all programmers using the local area network. All the files in the new version become visible simultaneously.

The *siphon* component is used to link repositories that are too far apart to be served by the same distributed file system. (In our case, the two repositories of interest are 6000 miles apart.) When a new version of a package is checked in at one repository, the siphon copies it to the other repository within a few hours. Again, all new files in a single package become visible simultaneously.

An earlier version of this system was coded with conventional RPC. The current version coded with network objects is distinctly simpler, for several reasons.

First, pickles and network streams simplified the interfaces. For example, to fetch a package, the old siphon enumerated the elements of the directory by repeated RPC calls; the new siphon obtains a linked structure of directory elements in one call. Also, the old siphon used multiple threads copying large buffers of data to send large files; the new siphon uses a network stream.

Second, third-party transfers eliminated an interface. The previous version of the siphon would pull a new version of a package from one of the source replicas, push it over the wide area network to a partner siphon at the other site, which would cache it on its disk and then push it to each of the destination replicas. Thus both a pull and a push interface were required. The new siphon transfers the object implementing the pull interface to its remote partner, which then pulls the files from the source replica directly. Thus the push interface was eliminated.

Third, we can take advantage of the ability to plug new transports into the system. Data compression is known to significantly increase bandwidth over wide area networks. Although we have not had need to do so, we could easily provide a subtype of `Transport.T` that automatically compresses and decompresses data. This would move the compression code out of the application and into a library where it could easily be reused.

In writing the Siphon system we deliberately stressed distributed garbage collection by performing no explicit deallocations. The results of this strategy were mixed. There were no serious memory leaks and garbage collection overhead was not a problem, but automatic reclamation was not as timely as we would have liked. The fundamental problem is that performance tradeoffs made in the local collector may not be appropriate for the distributed case. For example, it may be perfectly acceptable for the collector to delay reclaiming some small object, but if the object is a surrogate this can prevent the reclamation of an object that holds some important resource. We also found that the Modula-3 local collector occasionally fails to free unreachable objects because of its conservative strategy, and that this is more of a problem for distributed computations than for local ones. We conclude that for an application like the Siphon system that holds important resources like Unix file handles, it is necessary either to rewrite the local collector or to code the application to free resources explicitly.

6.2 Argo

Argo is a desktop telecollaboration system using audio, video, a shared whiteboard, and shared application windows to facilitate cooperation among multiple users who may be separated across long distances. A central function of Argo is conference control, which coordinates the sharing of the various media and tools to provide a coherent model of group collaboration. This shared state is held in a small special-purpose database that is implemented in terms of network objects.

The conference control server's database defines three types of objects: *users*, *conferences*, and *members*. A user object represents a human user of the system; a conference object represents an collaboration, such as a virtual conference room. The basic event is that users join and leave conferences. A member represents a {user, conf} pair and is created automatically when a user joins a conference. Each object in the database has a list of properties whose meaning is defined by client programs. General property lists were chosen instead of a predefined hierarchy of subtypes to increase independence among client programs.

The primary function of the server is to notify clients of events that occur in its database. This is done via callbacks. A client program registers a *handler* object and an *event filter*. When an event passes the filter, an appropriate callback method of the client's handler object is invoked, and is passed the relevant database object(s) that were involved in the event. For example, when a user joins a conference, the client programs involved in the conference are notified and can obtain the properties of the new user's object.

Callbacks like those employed in Argo are commonplace in non-distributed applications, and network objects extend this style to distributed programming. Nonetheless, transparent distributed invocation is not a panacea; distributed programs are inherently more complex than centralized ones. For example, callbacks from the Argo server to clients cannot be treated like local callbacks: the server must protect itself against clients that crash or are too slow, especially when locks are involved. Although good tools can hide many of the tiresome details of distributed programming, they do not yet eliminate the fundamental issues that must be faced in designing a robust distributed system.

The ease of defining, debugging and modifying the Argo conference control system and its protocol via network objects has been quite striking. Because of the leverage provided by Modula-3 and network objects, the entire conference control server implementation contains only 1400 lines of source code.

6.3 Obliq

Obliq is a lexically-scoped, untyped, interpreted language that supports distributed object-oriented computation. Obliq objects have state and are local to a site, but computations can roam over the network.

The characteristics of Modula-3 network objects had a major influence on Obliq, not just in the implementation, but also in the language design. All Obliq objects are implemented as network objects, so there is no artificial separation between local Obliq objects and those that may be remotely accessed. Also, all Obliq program variables are network objects, including global variables. Therefore, a remote computation can still access global state at the site at which it originated. Two elements of the network objects system were particularly useful in simplifying the Obliq implementation. Distributed garbage collection relieved concerns about space reclamation, and marshaling via pickles made it easy to transmit complex data structures such as the runtime representation of Obliq values.

6.4 Other work

Network objects are also in use in a system for continuous performance monitoring. The system provides a *telemonitoring* server which can be directed by the user to retrieve and process event logs from remote programs. The logs are communicated to the telemonitor via remote readers. Because a monitored program may disconnect from one telemonitor and reconnect to another, this application requires the ability to disconnect a network reader and reconnect the underlying data stream to a different process. This works, but requires extra logic in the application. In order to guarantee that no data is lost, the telemonitor must call the monitored program to indicate that it is disconnecting.

Our experience with the continuous monitoring project showed us that the design of reader and writer marshaling is trickier than it would at first appear. The semantics of marshaled streams are surprisingly difficult to specify and there are many design tradeoffs involved. We chose to give rather weak semantics, barring third-party marshaling of streams and specifying little about the state of a concrete stream after marshaling. Providing stronger guarantees would have had a high cost in either throughput or complexity.

We have also implemented secure network objects. A secure network object method can authenticate its caller, which allows security based on access control lists. A secure network object can also be passed to third parties and cannot be forged, which allows security based on capabilities. A client can use secure and ordinary insecure network objects together in the same application, incurring a performance penalty only for secure invocations or third party transfers of secure objects. Leendert van Doorn has implemented secure network objects[28], but the implementation has not yet been released or extensively used.

We learned three things from our secure network object implementation. First, the design of the secure system followed naturally from that of the insecure system. A large-scale redesign was not necessary. Second, because we use readers and writers for marshaling, it was easy to insert reader and writer subtypes that perform the cryptographic functions necessary for network security. Finally, because of our desire to make remote invocations transparent, we did not identify the caller via an implicit argument to the owner's method. Instead, we require callers to pass an explicit extra argument if they want to identify themselves. This argument is of a distinguished type that is marshaled specially.

Several users of network objects have noted our lack of support for object persistence. We note that Carsten Weich has recently added support for stable objects to the Modula-3 runtime system. He captures a stable snapshots of an object's state and then writes the arguments of update methods into a redo log using techniques quite similar to marshaling.

Providing support for stable concrete objects is not the whole story, however. In a distributed system, it can be valuable for a surrogate object to remain valid after restart of the owner's address space. We have not implemented this facility, although we believe that it would be straightforward to do so with a library built on top of the existing network object system.

7 Conclusion

The narrowest surrogate rule is flexible, but the associated type checking is dynamic rather than static, which has all the usual disadvantages. Because programs can be relinked and rerun at any time, it seems impossible to provide purely static type checking in a distributed environment. Dynamic checking imposes a burden of discipline upon programmers. The most common failure during the early stages of debugging a network objects application is a narrow fault (failure of a runtime type-check). For example, if a programmer forgets to link in stubs for a subtype `A` of `NetObj.T`, an import of an `A` object will succeed, but the resultant surrogate will have type `NetObj.T` and any attempt to `NARROW` it to an `A` object will fail.

Even an application that has been in service for a long time can crash with a narrow fault if some programmer carelessly changes a low-level interface and rebuilds another program with which the application communicates. Because of this danger, programmers should minimize external dependencies when defining a network object subtype. It is also important that the implementation of type fingerprinting not introduce spurious dependencies.

Programmers appreciate the narrowest surrogate rule, and more than one has asked for comparable flexibility in the case of ordinary objects. (If an attempt is made to unpickle an ordinary object into a program that does not contain the type of the object, an exception is raised.) But in this case liberality seems unsound. Suppose type `AB` is derived from `A`, and that we contrive to send a copy (rather than a reference) of an object of type `AB` into a program that knows the type `A` but not the type `AB`. One can imagine doing this either by ignoring the `B` data fields and methods, or by somehow holding them in reserve. In either case, the new program can operate on the A part of the state, for example by reading or writing data fields or by calling methods of A. However, there is no guarantee that these operations will be valid, since the original type AB may have overridden some of the methods of A; for example in order to accommodate a change in the meaning of the representation of the A fields. The narrowest surrogate rule seems sound only when objects are transmitted by reference.

The programmers that have used network objects have found the abstractions it offers to be simple yet powerful. By providing transparent remote invocation through Modula-3 objects, we eliminate many of the fussy details that make RPC programming tedious. Through the use of pickles, and by implementing third party network object transfers and marshaled abstract streams, we remove many restrictions about what can be marshaled, and we do so without increasing the complexity of generated stubs. The strength of our system comes not from proliferating features, but from carefully analyzing the requirements of distributed programming and designing a small set of general features to meet them.

Acknowledgements

We are grateful for help and suggestions from Bob Ayers, Andrew Black, John Ellis, David Evers, Rivka Ladin, Butler Lampson, Mark Manasse, and Garret Swart. We thank Luca Cardelli and Dave Redell for providing words describing the Obliq and Argo projects. Paul Leach, Sharon Perl, Allan Heydon, and Cynthia Hibbard made helpful comments on the presentation of this report.

References

[1] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: A technical review. *IEEE Trans. Software Engineering*, 11(1):43–59, 1985.

[2] Henri E. Bal, M. Frans Kaashoek, and Andy S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Software Engineering*, 18(3):190–205, 1992.

[3] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed garbage collection for Network Objects. Research report 116, Systems Research Center, Digital Equipment Corp., 1993.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure call. *ACM Trans. Computer Systems*, 2(1):39–59, 1984.

[5] Andrew P. Black and Yeshayahu Artsy. Implementing location independent invocation. *IEEE Trans. Parallel and Distributed Systems*, 1(1):107–119, 1990.

[6] Luca Cardelli. Obliq: A language with distributed scope. Research report 122, Systems Research Center, Digital Equipment Corp., 1994.

[7] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proc. 12th ACM Symposium on Operating System Principles*, pages 147–158, 1989.

[8] Graeme N. Dixon, Santosh K. Shrivastava, and Graham D. Parrington. Managing persistent objects in Arjuna: a system for reliable distribed computing. Technical report, Computing Laboratory, University of Newcastle upon Tyne, undated.

[9] Hania Gajewska, Jay Kistler, Mark S. Manasse, and David D. Redell. Argo: A system for distributed collaboration. In *Proc. ACM Multimedia '94*, pages 433–440, 1994.

[10] David K. Gifford and Nathan Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Trans. Computer Systems*, 6(3):258–283, 1988.

[11] Graham Hamilton, Michael L. Powell, and James G. Mitchell. Subcontract: A flexible base for distributed programming. Technical Report SMLI TR-93-13, Sun Microsystems Laboratories, 1993.

[12] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Trans. Programming Languages and Systems*, 4(4):527–551, 1982.

[13] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some useful Modula-3 interfaces. Research report 113, Systems Research Center, Digital Equipment Corp., 1993.

[14] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Trans. Computer Systems*, 6(1):109–133, 1988.

[15] K. Rustan M. Leino. Extensions to an object-oriented programming language for programming fine-grain multicomputers. Technical report CS-TR-92-26, California Institute of Technology, 1992.

[16] Barbara Liskov. Distributed programming in Argus. *CACM*, 31(3):300–312, 1988.

[17] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proc. 11th ACM Symposium on Operating System Principles*, pages 111–122, 1987.

[18] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1991. ISBN 0-13-590464-1.

[19] Object Management Group. *Common Object Request Broker: Architecture and Specification*, 1991. OMG Document 91.12.1.

[20] Open Software Foundation. *DCE Application Development Reference, volume 1, revision 1.0*, 1991.

[21] Francis Prusker and Edward Wobber. The Siphon: Managing distant replicated repositories. Research report 7, Paris Research Laboratory, Digital Equipment Corp., 1991.

[22] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. Research report 59, Systems Research Center, Digital Equipment Corp., 1990.

[23] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Trans. Computer Systems*, 8(1):1–17, 1990.

[24] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *IEEE International Conference on Distributed Computer Systems*, pages 198–204, 1986.

[25] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–337, 1989.

[26] J. E. Stoy and C. Strachey. OS6—an experimental operating system for a small computer. Part 2: input/output and filing system. *The Computer Journal*, 15(3):195–203, 1972.

[27] Andrew S. Tannenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *CACM*, 33(12):46–63, 1990.

[28] Leendert van Doorn, Martín Abadi, Mike Burrows, and Edward Wobber. Secure Network Objects. Research report IR-385, Vrije Universiteit, Amsterdam, 1995.

Index