115

Network Objects

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

February 28, 1994

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

Authors' Abstract

A network object is an object whose methods can be invoked over a network. This report describes the design and implementation of a network objects system for Modula-3. The system is novel for its overall simplicity. The report includes a thorough description of realistic marshaling algorithms for network objects, precise informal specifications of the major internal interfaces, preliminary experience, and performance results.

Contents

Preface

This report describes the design and implementation of a network objects system, which allows you to write programs that communicate over a network, while hiding the messy details of network programming. Network objects provide functionality similar to remote procedure call (RPC) [3], but they are more general and easier to use. The system is implemented in Modula-3 [17].

The report includes an overview of the system and specifications for the main interfaces. If you have never written a distributed program in your life, you can use this report to learn how; if you are experienced in the field, you can use this report to learn how network objects are different from other similar systems and from traditional RPC.

Section 1 provides an overview of the system, including descriptions of the major modules and the interfaces between them. A previous version of this section appeared in the Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, Asheville, North Carolina, December, 1993. The remaining sections of the report provide a reference manual. They list the actual Modula-3 interfaces together with precise informal specifications.

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 [15], Eden [1], and early work of Shapiro's [23], 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*.

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.

In addition to providing a design rationale, this report presents 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.

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 (powerful marshaling, strong type-checking, distributed garbage collection, efficient and convenient access to streams), and we omitted more complex or speculative features (transactions, object migration, distributed shared memory). Also, we organized the implementation around a small number of quite simple interfaces, each of which is described in this report. Finally, we believe we have done this without compromising performance: we have not worked hard on performance, but we believe our design is compatible with a high-performance implementation.

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. Marshaling automatically handles format differences in the two programs (for example, different byte orders for representing integers).

It is difficult to provide fully general marshaling code in a satisfactory way. Existing systems fail in one or more of the following ways. Some apply restrictions to the types that can be marshaled, typically prohibiting linked, cyclic or graph-structured values. Some generate elaborate code for almost any data type, but the resulting stub modules are excessively large. Some handle a lot of data types, but the marshaling code is excessively inefficient. We believe we have achieved a better compromise here by the use of a general-purpose mechanism called *pickles* [5]. This uses 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 (for efficiency), but calls the pickle package for complicated types (for compactness of stub code). We believe our pickling machinery's performance is within a factor of two of the performance of mechanically generated in-line code.

Since we marshal by pickling and the pickle package will handle any reference type, it's possible to marshal arguments or results that are objects. There are two cases. If the object being marshaled (by pickling) is a network object, it is passed as an object reference (as described later). Alternatively, if the object being marshaled is not a network object, it is marshaled by copying the entire object's value. This provides a form of object mobility that is satisfactory for many purposes. For example, a system like Hermes [6], though designed for mobile objects, could be implemented straightforwardly with our mechanisms.

Inter-process byte streams are more convenient and efficient than RPC for transferring large amounts of unstructured data, as critics of RPC have often pointed out. We have therefore provided special marshaling support for Modula-3's standard stream types (readers and writers). 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.

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 [19], 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.

1.1 Related work

We have built closely on the ideas of Emerald [13] and SOS [24]; 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. However, pickles make it easy to transfer non-network objects by copying, which offers some of the benefits of mobility while avoiding the costs.

We also have used some of the ideas of the conventional (not object-oriented) DCE RPC system [19, Part 2]. Network objects simultaneously generalize DCE bindings and context handles.

Systems like Orca [2] and Amber [8] 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 [14], but we will not describe his work here.)

Systems like Argus [15, 16] and Arjuna [9] 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 [18]. 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 [17].

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 (so-called *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. The test may be performed by the application code (if, for example, the declared return type is a generic network object, but the expected return type is more specific), by the stubs (if the declared return type is specific), or by the generic unpickling code (if a linked data structure includes a field whose type is a specific subtype of network object). Notice that the third case imposes 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.

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 fact. 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. 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 provision for notifying that 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 computations, 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. The 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 causes the thread to raise `NetObj.Error` 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.

1.5 Implementation

This subsection describes the structure of our implementation. The description is reasonably detailed, but we have made a few simplifications here and there. Sections 2 and 3 of this report list the actual system interfaces.

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 and the ability to test the type of an object at runtime, to allocate an object given a code for its type, to find the code for the direct supertype given the code for the type, and

to determine at runtime the sizes and types of the fields of an object, given the type of the object;

2. threads (lightweight processes);

3. some form of reliable, inter-address-space communication;

4. garbage collection together with a hook for calling a cleanup routine when an object is garbage collected (or explicitly freed);

5. a method of communicating object typecodes from one address space to another; and finally,

6. object-oriented buffered streams.

We will elaborate on the last two items.

The numerical typecodes assigned by the compiler and linker are unique within a given address space, but not across address spaces. Compiler support is required to solve this problem. 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.

A buffered stream is an object type 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 [25]. In Modula-3 they are called *readers* and *writers*, and are described in Chapter 6 of [17].

Garbage collection. Our system includes network wide reference counting garbage collection. For each exported 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, a procedure is registered with the local garbage collector to be called when the surrogate is collected. This procedure makes an RPC call to the owner to remove itself from the dirty set. When the dirty set becomes empty, the runtime discards the retained pointer, allowing the owner's local garbage collector to reclaim the object if no local references remain.

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

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 acknowledgment to any result message that contains a network object: the procedure that executes the call in the owner blocks waiting for the acknowledgment, 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 mis-interpreted 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. 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 are increasing from any 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.

The companion report [4] presents the details of the collection algorithm and a proof of its correctness.

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

A `Transport.T` is an object that generates and manages connections between address spaces. Different subtypes use different communication methods. 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 `tr.fromEndpoint(ep)` 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 [3]. That is, in spite of its name, 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 in Section 3.5.

Basic representations. 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` for 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.)

If a non-surrogate network object is present in the object table, we say that it is *exported*; otherwise it is *unexported*. The concrete representation of a network object includes a `state` field that records whether the network object is a surrogate, and, if not, whether the object is exported:

```
TYPE State = {Surrogate, Exported, Unexported};

REVEAL
  NetObj.T = OBJECT
    state: State := Unexported;
    wrep: WireRep;
    loc: Location;
    disp: Dispatcher
  END;

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

(The `REVEAL` statement specifies the concrete representation of the opaque type `NetObj.T`. The `state` field is given a default value.)

If `obj.state` is `Exported` or `Surrogate`, then `obj.wrep` is the wire representation of the object.

If `obj.state` is `Surrogate` then `obj.loc` is a `Location` that generates connections to the owner's address space at `obj.wrep.sp`, and `obj.disp` is unused. If `obj.state` is `Exported`, 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 and sends the result over `c`.

Remote invocation. To illustrate the steps in a remote method invocation we use the type `FS.Server` defined in Section 1.3, 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 presented in Section 3. (Actually, `StubLib.OutRef` subsumes both `OutNetObj` and `OutText`, but we ignore that here.)

The method 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 `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: FS.File;
      BEGIN
        res := obj.open(n);
        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 (see Section 3) as well as type code, to make it easy for a program to support multiple protocol versions.)

Marshaling network objects. The call `OutNetObj(c, obj)` writes to the connection `c` the wire representation of the reference `obj`:

```
PROCEDURE OutNetObj(c: StubLib.Conn; obj: NetObj.T) =
  BEGIN
    IF obj = NIL THEN
      OutWireRep(c, (-1,-1));
    ELSE
      IF obj.state = Unexported THEN
        VAR
          i := NewObjID();
        BEGIN
          obj.wrep := (SelfID(), i);
          objtbl[obj.wrep] := obj;
          obj.type := Exported;
          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 = 0 THEN
      RETURN NIL
    ELSIF wrep IN domain(objtbl) THEN
      RETURN objtbl[wrep]
```

```
    ELSE
      RETURN NewSurrogate(wrep, c)
    END
  END InNetObj;
```

The call `NewSurrogate(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`.)

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

```
PROCEDURE NewSurrogate(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.state := Surrogate;
    objtbl[wrep] := res;
    RETURN res
  END NewSurrogate;
```

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. For any space sp, names[sp] is the sequence of endpoints for sp recognized by sp's transports. (We write SEQ[T] to denote the type of sequences of elements of type T.) The tr sequence is filled in at initialization time, with one entry for each transport linked into the program, and is constant thereafter.

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 that is 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 `NewSurrogate`. 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. We have also omitted the code for relaying exceptions raised in the owner to the client, and for relaying thread alerts from the client to the owner.

Marshaling streams. 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 `MarshalObject(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` obtains a usable connection with `nc := vs.loc.new()` and 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.

Pickling and marshaling. The default behavior of the pickle package isn't satisfactory for all types. Most often this arises because the concrete representation of an abstract type (for example, a mutex or stream) isn't the 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 implementor of an abstract data type would specify such a custom procedure if the type's values weren't transferable by straightforward copying. The details of this mechanism are beyond the scope of the present report.

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 rule that network objects are always transmitted by reference applies to marshaling only. In other situations it is important to be able to pickle network objects by copying; for example, when writing a checkpoint to a disk file. To accommodate this need, the procedures registered for pickling and unpickling network objects perform a runtime test of the type of their stream argument. If the stream is of the distinguished subtype declared in the transport interface, then they use the marshaling algorithms described above; otherwise they copy the data fields, as for an ordinary object.

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 ID 0, of a known type, provided by the network object runtime. 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 and exports it under ID 0.

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 procedure `NetObj.Locate(nd)`, which appeared in our example in Section 1.3, and is specified in more detail in Section 2.1, simply forges a surrogate for the special object of the agent on the node `nd`. The procedures `NetObj.Import` and `NetObj.Export` use the `get` and `put` methods of that special object.

1.6 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 done much performance analysis or optimization, but we have measured the times for some basic operations. The following numbers were taken using Digital workstations equipped with MIPS R3000 processors (25 Specmarks) running Ultrix and communicating over a 100 megabit/sec AN1 network [21]. Here are the numbers:

Null call	3109 usec/call
Ten integer args	3278 usec/call
One object arg (s)	3417 usec/call
One object arg (c)	3485 usec/call
One object arg (c,d)	8785 usec/call
One object result (s)	3758 usec/call
One object result (c)	3851 usec/call
One object result (c,d)	8747 usec/call
Reader test	2910 kilobytes/sec
Writer test	2876 kilobytes/sec

(c) concrete object marshaled
(s) surrogate object marshaled
(d) dirty call required

For comparison, on the same configuration it takes 1.6 milliseconds for a C program to echo a TCP packet from user space to user space. The remaining 1.5 milliseconds for the null call is spent in two Modula-3 user space context switches (which together add some 170 microseconds of latency to the null call) plus the cost of marshaling and unmarshaling the object whose null method is being called.

The ten integer argument test shows that the incremental cost of an integer argument is about 17 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 was marshaled as a surrogate and unmarshaled as a concrete object. A `(c)` indicates that the argument or result was marshaled as a concrete object and unmarshaled as a surrogate. A `(d)` indicates that a dirty call was required.

The incremental cost of a surrogate network object argument that does not lead to a dirty call is roughly 210 microseconds. A concrete network object argument is somewhat more expensive. If a dirty call is required, there is an additional cost of about 5 milliseconds.

Network object results are more expensive than arguments, because of the acknowledgement that must be sent when the result message contains a network object (see page 14). In the tests that do not involve dirty calls, this cost shows up as a difference of approximately 350 microseconds, but in the tests that do involve dirty calls the cost seems to be lost in the noise.

The last two tests give the throughput of network streams. For comparison, the throughput of a raw TCP stream using a C program is about

3400 kilobytes/sec; the difference is largely the cost of the Modula-3 user space thread switch required for network streams. (Neither raw TCP connections nor our streams built over them succeed in using much of the 100 megabits supplied by the network, for reasons that are outside the scope of this report.)

The purpose of our project was to find an attractive design, not to optimize performance, and our numbers reflect this. (They also reflect the costs of Modula-3 runtime checks and the fact that the native code Modula-3 compiler is still in alpha test.) We are confident that the techniques that have made RPC systems fast (at least in research laboratories) could be applied effectively to our system.

1.7 Experience

Our system has been working for a few months. Several projects are building on the system, including an experimental telecollaboration system and a lightweight distributed application builder based on Luca Cardelli's language Obliq [7]. Perhaps the most significant completed projects are the packagetool and the siphon.

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* 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.

The previous version of the siphon was coded with conventional RPC. The new version 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, although we have not done so yet, we expect to take advantage of the ability to plug new transports into the system easily. We would like to use data compression to reduce the number of bytes sent over the wide area network. We plan to do this by providing a subtype of `Transport.T` that automatically compresses and decompresses data. Thus the compression code will be moved out of the application and into a library where it can easily be reused.

1.8 Conclusions

Our system is simple to use and to implement, because objects have eliminated many of the fussy details about bindings, subtyping allows for interface evolution, there is a clean interface between the transports and the runtime proper, and pickles have eliminated many of the restrictions about what can be marshaled.

The performance of our system is already adequate for many purposes. It is competitive with the performance of commercially available RPC systems [19], and we believe our measurements indicate that our design does not prevent the sort of improvements that result in the RPC performance levels reported by various research groups [22, 26].

2 Programmer's reference manual

This section describes the `NetObj`, `NetStream`, and `NetObjNotifier` interfaces, as well as the stub generator. This is enough information for all but the most specialized clients of network objects.

The interfaces described here and in the following sections depend on a few local-level facilities from the SRC Modula-3 runtime library [12, 17]. 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.

2.1 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.

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 standard 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 possible failures in a remote method invocation. Every remote method should therefore include `NetObj.Error` 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. The subsequent atoms may provide additional implementation-dependent details about the failure. Here are the interpretations of the first atom:

The atom `CommFailure` indicates communication failure, which might be network failure or a crash on a remote machine. `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.) Any other atoms indicate errors inside the network object runtime.

2.2 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 freely as parameters and as results of remote method invocation. For any unmarshaled subtypes of `Rd.T`, the concrete reader is replaced by a surrogate reader whose source is the remaining source of the concrete reader. Similarly for subtypes of `Wr.T`, the concrete writer is replaced by a surrogate writer whose target is the target of the concrete writer. Surrogate streams are free-standing entities, valid beyond the scope of the remote call which 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. Undefined behaviour will result if the same reader or writer is marshaled to more than one client, or is used concurrently at both surrogate and concrete instances.

Calling `Wr.Flush` on a surrogate writer flushes all outstanding data to the remote writer and flushes the remote writer. Calling `Wr.Close` flushes and then closes both the surrogate and the remote 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 reponsible for closing the concrete reader. The `WeakRef` interface can be used to register a GC finalization procedure for this purpose.

Concrete writers can be serially reused by different address spaces assuming that each space flushes its surrogate before relinquishing it. If you want to do this, use the `ReleaseWr` procedure to free the network resources associated with discarded surrogates.

Concrete readers can also be serially reused, and again you should free the network resources associated with discarded surrogates, in this case using `ReleaseRd`. However, there is no guarantee that the current position of a concrete reader equals that of its surrogate; read-ahead is employed at the surrogate for efficiency. Therefore, in order to reuse a concrete reader effectively, you will have to reset it to some known position after releasing the surrogate.

```
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`, close `rd`, but do not close the concrete target of `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 target of `wr`. This procedure is a no-op if `wr` is not a surrogate.

```
END NetStream.
```

2.3 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);
```

Register a `NotifierClosure` for `obj`. Multiple closures for the same object are permissible. If `obj` is not a surrogate object then `AddNotifier` has no effect.

```
END NetObjNotifier.
```

The `notify` method of a `NotifierClosure` is invoked when the concrete object corresponding to the surrogate `obj` becomes inaccessible. The procedure `AddNotifier` must have been called to enable this notification. 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.

2.4 The stub generator

The stub generator is a program that generates stubs for Modula-3 network object types.

Restrictions. 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:

- Stubs can only be generated for named types. For example, suppose interface `Foo` contains the following declarations:

  ```
  TYPE
    A = NetObj.T OBJECT METHODS x() END;
    B = RECORD b: NetObj.T OBJECT METHODS y() END END;
  VAR c: NetObj.T OBJECT METHODS z() END;
  ```

  If the stub generator is invoked over the interface `Foo`, it will produce stubs only for `A`, and not for the anonymous subtypes used to declare `B.b` and `c`.

- 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`. The `-i` option (described below) allows the programmer to expose type revelations for objects that would be opaque during normal compilation.

- The methods of the type and its supertypes must have distinct names.

- A method argument may not be of type `PROCEDURE` or have a component of type `PROCEDURE`. (A network object with an appropriate method can always be sent instead of a procedure.)

- A method may not have a `RAISES ANY` clause.
- A valid network object type must be pure, that is it cannot contain data fields, either in its declaration or in a revelation.
- Branded types that are used, directly or indirectly, in a network object declaration must have explicit brands. This is necessary so that programs linked separately will have the same brands for the types. The stub generator will not generate stubs for a network object type if it can detect that this rule is violated. If an argument or result of such a type is supplied at runtime, there is likely to be a `NARROW` fault if the receiver tries to `NARROW` the value to the branded type.

Stub versions. Any change 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. Thus, multiple versions of the stub generator may sometimes exist at the same time. The command `stubgenv<n>` invokes version `n` of the stub generator, if that version is currently available. The command `stubgen` invokes the most recent version. At present only version `1` exists.

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. When version `<n+1>` is released, applications can be linked with stubs for both versions `<n>` and `<n+1>`. Re-linked instances of the application will communicate with each other using the new protocol, and with other instances of the application using the older protocol. When all instances of the application include the new protocols, it is safe to relink using only version `<n+1>`.

Arguments and results. 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.

Invoking the stub generator. Most users need not invoke the stub generator directly, because the standard SRC Modula-3 tool `m3build` provides the `netobj` command that invokes the stub generator and compiler automatically. However, this section sketches the command line interface to the stub generator for completeness.

The stubs for a network object of type `Test.T` consist of two files:

```
Test_T_v<n>.i3    and    Test_T_v<n>.m3
```

The notation `<n>` indicates the version number of the stub generator that produced the files. These files must be compiled and linked with any application that uses objects of type `Test.T` as network objects. Failure to link in the stubs will most likely result in a run-time `NARROW` fault the first time an object of type `Test.T` is received as an argument or result.

The arguments to `stubgen` determine the set of types for which stubs are produced. The command

```
stubgen -i Intf1 Intf2 Intf3
```

produces stubs for all valid network object types declared in the listed interfaces. The command

```
stubgen -sno Intf1.T1 Intf2.T2
```

produces stubs for the named network object types. Thus if the only valid network object types declared in `Intf1` are `T1` and `T2`, the following two commands are equivalent:

```
stubgen -i Intf1
stubgen -sno Intf1.T1 Intf1.T2
```

When `stubgen` is invoked with both interface and network object arguments, it produces stubs for the named network objects, making use of any relevant revelations in the given interfaces. For example, if `T` is defined opaquely in the interface `Opaque`, with a revelation in the interface `Reveal`, its stubs can be produced with the command:

```
stubgen -sno Opaque.T -i Reveal
```

Note that the command would not succeed without the `-i` argument. As explained above, stubs for an opaque network object type can only be generated when its revelations are available to the stub generator.

3 Internal system interfaces

In this section we present the main internal systems interfaces. The typical programmer using network objects has no need to read them. But if you are writing your own stub generator, or hand-coding stubs for some particular network object type, or adding a new transport to the system, or if you are just curious, here they are.

3.1 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 automatic conversion is impossible, the the exception NetObj.Error(NetObj.UnsupportedDataRep) is raised.

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. `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` 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 acknowledgment, that is, whether the result contained any network objects. If an acknowledgment 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) =
   <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 on `c`.

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.

For any subtypes of `NetObj.T` in the pickled datatype, a surrogate network object is substituted for the concrete object. The runtime guarantees that within a single address space, only one surrogate for any single concrete object will exist at any given time.

For any unmarshaled subtypes of `Rd.T`, the concrete reader is replaced by a surrogate reader whose source is the remaining source of the concrete reader. Similarly for subtypes of `Wr.T`, the concrete writer is replaced by a surrogate writer whose target is the target of the concrete writer. The operations on surrogate streams are described in `NetStream.i3`.

Marshaling of generic data. The following procedures are made available to permit the generic marshaling of various primitive data types.

```
PROCEDURE OutChars(
    c: Conn; READONLY chars: ARRAY OF CHAR)
    RAISES {Wr.Failure, Thread.Alerted};
```

Marshal a char array in native format.

```
PROCEDURE OutBytes(
    c: Conn; READONLY bytes: ARRAY OF Byte8)
    RAISES {Wr.Failure, Thread.Alerted};
```

Marshal a byte array.

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

Marshal an integer in native format.

```
PROCEDURE OutInt32(c: Conn; i: Int32)
    RAISES {Wr.Failure, Thread.Alerted};
```

Marshal a 32-bit integer in native format.

```
PROCEDURE OutByte(c: Conn; i: Byte8)
    RAISES {Wr.Failure, Thread.Alerted};
```

Marshal a byte.

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

Marshal a boolean value.

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

Marshal a real in native format.

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

Marshal a longreal in native format.

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

Marshal an extended in native format.

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

Marshal a cardinal in native format.

The following procedures are provided in support of generic unmarshaling of data. In all cases, `rep` indicates the encoding of the incoming data. These procedures could be replaced by inline unmarshaling code whenever the relevant elements of `rep` match the corresponding elements of `NativeRep`.

```
PROCEDURE InChars(
    c: Conn; rep: DataRep;
    VAR chars: ARRAY OF CHAR)
    RAISES {NetObj.Error, Rd.Failure, Thread.Alerted};
```

Unmarshal a char array of length `NUMBER(chars)`.

```
PROCEDURE InBytes(
    c: Conn; VAR bytes: ARRAY OF Byte8)
    RAISES {NetObj.Error, Rd.Failure, Thread.Alerted};
```

Unmarshal a byte array of length `NUMBER(bytes)`.

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

Unmarshal an integer, checking that its value is in `[min..max]`.

```
PROCEDURE InInt32(
    c: Conn;
    rep: DataRep;
    min := FIRST(Int32);
```

```
    max := LAST(Int32)): Int32
    RAISES {NetObj.Error, Rd.Failure, Thread.Alerted};
```

Unmarshal a 32-bit integer, checking that its value is in `[min..max]`.

```
PROCEDURE InByte(
    c: Conn;
    max := LAST(Byte8)): Byte8
    RAISES {NetObj.Error, Rd.Failure, Thread.Alerted};
```

Unmarshal a byte, checking that its value is in `[0..max]`.

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

Unmarshal a boolean value.

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

Unmarshal a real value.

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

Unmarshal a longreal value.

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

Unmarshal an extended value.

```
PROCEDURE InCardinal(
    c: Conn; rep: DataRep;
    lim: CARDINAL := LAST(CARDINAL)): CARDINAL
    RAISES {NetObj.Error, Rd.Failure, Thread.Alerted};
```

Unmarshal a cardinal, checking that its value is in `[0..lim]`.

Here are two procedures for raising `NetObj` exceptions conveniently:

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

Raise `NetObj.Error(AtomList.List1(UnmarshalFailure))`.

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

Raise `NetObj.Error(AtomList.Cons(NetObj.CommFailure, ec))`.

```
END StubLib.
```

3.2 An example stub

This subsection illustrates the use of the `StubLib` interface by presenting hand-generated stubs 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};
     put(key: TEXT; value: 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 `Thread.Alerted` is not propagated 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, Put };  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;
    put := SurrogatePut;
  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;

PROCEDURE SurrogatePut (t: StubT; key: TEXT; value: TEXT)
    RAISES {Invalid, NetObj.Error, Thread.Alerted} =
  VAR reuse := FALSE;
      rep: StubLib.DataRep;
      c: StubLib.Conn;
  BEGIN
    TRY
      c := StubLib.StartCall(t, StubVersion);
      TRY
        StubLib.OutInt32(c, ORD(P.Put));
        StubLib.OutRef(c, key);
        StubLib.OutRef(c, value);
        rep := StubLib.AwaitResult(c);
        CASE StubLib.InInt32(c, rep) OF
        | ORD(R.OK) =>
            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;
  END SurrogatePut;
```

`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);
      | ORD(P.Put) => PutStub(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;
```

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

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.
```

3.3 StubConn interface

A `StubLib.Conn` represents a bi-directional 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.

Clients can use this interface to bypass the procedures in the `StubLib` interface and marshal and unmarshal arguments using inline code, for example to write directly to the underlying writer. In doing so, beware that all parameters of type `Conn` must be passed to the procedures in the `StubLib` interface with both the reader and writer components unlocked.

```
INTERFACE StubConn;

IMPORT MsgRd, MsgWr, StubLib;

REVEAL StubLib.Conn <: Public;

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

END StubConn.
```

The writer's `nextMsg` method must leave room for at least 24 bytes in the buffer. For any `rd` used to read messages written on `wr`, if a (partial) message is written to `wr` with length `len`, then `rd.nextMsg` must leave at least `MIN(24, len)` bytes in the buffer.

The reader and writer components of `Public` are both unmonitored: clients must not access them from multiple threads concurrently.

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. The specifications of these interfaces are in [17], chapter 6. Importing the interfaces will allow you to write stubs that operate directly on the character arrays `c.wr.buff` and `c.rd.buff`.

3.4 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 represent these message streams.

Message readers. A `MsgRd.T` is a reader that presents the abstraction of a stream of messages. A message is a (possibly zero length) sequence of bytes terminated by an end of message marker. The reader is initially positioned to the start of the first message. When the end of message marker is encountered, 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 will be 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.
```

The `nextMsg` method of a `MsgRd.T` advances the reader to the next message. A return value of `TRUE` indicates the presence of a new message. A return value of `FALSE` indicates that no next message is present, and that the end of the stream of messages has been reached.

`MsgRd.T` readers maintain the following invariant about the alignment of reader data as it appears in memory: for all `i`,

```
(ADR(buffer byte for src[i]) MOD 8) = (i MOD 8).
```

That is, message reader data is 8-byte aligned as it appears in the reader buffer. This allows optimized stubs to read scalar values from the buffer efficiently.

After the `nextMsg` method returns `TRUE`, the abstract reader state is as follows:

`len(rd)`	is the length of the new message
`src(rd)`	is the contents of the new message
`cur(rd)`	is zero
`avail(rd)`	is unspecified
`closed(rd)`	is unchanged
`seekable(rd)`	is unchanged
`intermittent(rd)`	is unchanged

Message writers. A `MsgWr.T` is a writer that presents the abstraction of a stream of messages. A message is a (possibly zero length) sequence of bytes terminated by an end of message marker. The writer is initially positioned to the start of the first message. The `nextMsg` method 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` will flush the current buffer to the abstract writer target, but will 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 will be 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.
```

The `nextMsg` method ends the current message and starts the next.

`MsgWr.T` writers maintain the following invariant about the alignment of writer data in memory: for all `i`,

```
ADR(buffer byte for target[i]) MOD 8 = i MOD 8.
```

That is, message writer data is 8-byte aligned as it appears in the writer buffer. This allows optimized stubs to write scalar values to the buffer efficiently.

After the `nextMsg` method returns, the abstract writer state is as follows:

`len(wr)`	is zero
`c(wr)`	is empty
`cur(wr)`	is zero
`target(wr)`	is the new message, which is initially empty
`closed(wr)`	is unchanged
`seekable(wr)`	is unchanged
`buffered(wr)`	is unchanged

3.5 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;
  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;

TYPE Endpoint = TEXT;
```

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

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

END Transport.
```

The main ideas in the interface were described in Section 1.5, page 15. 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.

See Section 1.5 for an overall discussion of these concepts. 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 a program instance `P` calls `tr1.toEndpoint()`, producing an endpoint `ep`, then the call `tr.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, as explained in Section 1.5, page 18. Ordinarily a transport implementation will not need to override this method. At the entry to `tr.serviceCall(conn)`, `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 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. By calling `loc.new`, the program acquires the obligation to 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 instance. 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.

Acknowledgments

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. Paul Leach, Sharon Perl, and Cynthia Hibbard made helpful comments on 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. on Software Engineering* 11(1), January 1985.

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

[3] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computer Systems* 2(1), February 1984, pp. 39–59.

[4] 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., December 1993.

[5] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Pickles: a system for automatic serialization of typed values. Systems Research Center, Digital Equipment Corp., in preparation.

[6] Andrew P. Black and Yeshayahu Artsy. Implementing Location Independent Invocation. *IEEE Trans. on Parallel and Distributed Systems* 1(1), January 1990.

[7] Luca Cardelli. Obliq: A lightweight language for network objects. User's manual, Systems Research Center, Digital Equipment Corp., 1994. Available by anonymous FTP from gatekeeper.pa.dec.com as pub/DEC/Modula-3/contrib/Obliq.ps.

[8] 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. *Proc. of the Twelfth ACM Symposium on Operating System Principles*, 1989, pp. 147–158.

[9] 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.

[10] David K. Gifford and Nathan Glasser. Remote Pipes and Procedures for Efficient Distributed Communication. *ACM Transactions of Computer Systems* 6(3), 1988, pp. 258–283.

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

[12] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some Useful Modula-3 Interfaces. Research report 113, Systems Research Center, Digital Equipment Corp., December 1993.

[13] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems* 6(1), February 1988, pp. 109–33.

[14] K. Rustan M. Leino. Extensions to an object-oriented programming language for programming fine-grain multicomputers. Caltech Technical Report CS-TR-93-26, 1992.

[15] Barbara Liskov. Distributed Programming in Argus. *CACM* 31(3), March 1988, pp. 300-12.

[16] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. *Proc. of the Eleventh ACM Symposium on Operating System Principles*, 1987, pp. 111–122.

[17] Greg Nelson, ed. *Systems Programming With Modula-3.* Prentice Hall Series in Innovative Technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991. ISBN 0-13-59046-1.

[18] Object Management Group. Common Object Request Broker Architecture and Specification. OMG Document Number 91.12.1.

[19] Open Software Foundation. DCE Application Development Reference, volume 1, revision 1.0. Threads, Remote Procedure Call, Directory Service. 11 Cambridge Center, Cambridge, MA 02141. 1991.

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

[21] 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 with point-to-point links. Research report 59, Systems Research Center, Digital Equipment Corp., April 1990.

[22] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Trans. Comp. Sys.* 8, 1, Feb. 1990, 1–17

[23] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. International Conference on Distributed Computer Systems, IEEE, May 1986.

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

[25] 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), 1972.

[26] A. Tanenbaum et al. Experiences with the Amoeba distributed operating system. *Comm. ACM* 33(12), December 1990.

Index