Network Objects

Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber
Digital Systems Research Center

Abstract

A network object is an object whose methods can be invoked over a network. This paper describes the design, implementation, and early experience with a network objects system for Modula-3. The system is novel for its overall simplicity. The paper includes a thorough description of realistic marshaling algorithms for network objects.

1 Introduction

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 [12], Eden [1], and early work of Shapiro's [18], 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?

This paper contributes a data point for these discussions by describing a network objects system we have recently implemented for Modula-3. In addition to providing a design rationale, we present 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, 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 paper. 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

SIGOPS '93/12/93/N.C., USA

of a general-purpose mechanism called `Pickles`. 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 [5], 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 addressed this issue by providing 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 in a way that minimizes data copies: for both the surrogate stream and the real stream, data is transferred between the stream buffer and the kernel via direct calls to `read` and `write`. An important feature of this design is that the stream data is not communicated via RPC, but by the underlying transport-specific communication. This facility is analogous to the remote pipes of DCE RPC [17], 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 [8], but is simpler and more transparent.

We also provide network-wide reference-counting garbage collection.

2 Related work

We have built closely on the ideas of Emerald [10] and SOS [19], and perhaps our main contribution has been to select and simplify the most essential features of these systems. An 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 (non-object-oriented) DCE RPC system [17, Part 2]. Network objects simultaneously generalize DCE bindings and context handles.

Systems like Orca [2] and Amber [6] aim at using objects to obtain performance improvements on a multiprocessor. We hope that our network object 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 [11], but we will not describe his work here.)

Systems like Argus [12, 13] and Arjuna [7] are like network objects in that they aim to support the programming of reliable distributed services; they differ by providing much 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 [9]. 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 [16]. 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.

3 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 [14].

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.

To implement network objects, the reference in the client program actually points to a *surrogate 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 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 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 is 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, the unpickler must check that the surrogate produced is legal at that position in the data structure).

4 Examples

To make these ideas more concrete, we will present some examples 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.
```

(Translation from Modula-3: The interface above 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:

```
MODULE Server EXPORTS Main;

IMPORT NetObj, FS, Time;

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.LocateHost("server"));
  f := s.open("/usr/dict/words");

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

The call `NetObj.LocateHost(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 `LocateHost` are described further in the section below on bootstrapping.

The client program makes the remote method calls `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 non-object-oriented 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;

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

Subtyping makes 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.

As a final example, a network object imported into a program that has no stubs linked into it at all 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.

5 Implementation

In this section we we will try to describe our system in sufficient detail to guide anyone who might want to reimplement it.

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 inter-address space communication, whether by reliable streams or unreliable datagrams;
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 in an address-space independent way. 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 [22]. In Modula-3 they are called *readers* and *writers*, and are described in Chapter 6 of [14].

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 dispatcher procedure blocks waiting for the acknowledgment, with the reference to the object on its stack, protected from its garbage collector. This 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. The potential for this error seems to be an unavoidable consequence of unreliability communication together with a desire to avoid storage leaks in long-running servers. Because we never reuse object IDs, we can detect this error 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 `CallFailed`. 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 paper [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.)

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 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 their connections. If creating connections is expensive, then the transport can cache them. If maintaining idle connections is expensive, the transport can close them. If both are expensive, as is often the case, the transport can cache idle connections for a limited amount of time.

(It is perfectly possible to implement a class of connection that communicates with datagrams according to a protocol that makes idle connections essentially free—see [3]. That is, in spite of its name, the type `Connection` need not be connection-oriented in the standard sense of the word.)

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, the connections generated by `c.loc` have the same direction as `c`.

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;
    wr: WireRep;
    loc: Location;
    disp: Dispatcher
  END;

TYPE Dispatcher =
  PROC(c: Connection; o: NetObj.T)
```

(The `REVEAL` statements 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.wr` is the wire representation of the object.

If `obj.state = Surrogate` then `obj.loc` generates connections to the owner's address space, and `obj.disp` is unused. If `obj.state = Exported`, then `obj.disp` is the dispatcher procedure for the object, and `obj.loc` is unused. The dispatcher 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 above, with a single method `open`. The corresponding stub-generated surrogate type declaration looks like this:

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

SrgOpen(ob: SrgSvr; n: TEXT): FS.File =
  VAR
    c := ob.loc.new();
    res: FS.File;
  BEGIN
    MarshalNetObj(ob, c);
    MarshalInt(0, c);
    MarshalText(n, c);
    Flush(c.wr);
    res := UnmarshalNetObj(c);
    ob.loc.free(c);
    RETURN res
  END;
```

We take for granted procedures for marshaling basic types. Procedures for marshaling network objects are described in the next subsection. 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 `UnmarshalNetObj` were not of type `FS.File` (as for example 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
  ob := UnmarshalNetObj(c);
BEGIN
  ob.disp(c, ob)
END;
```

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

```
SvrDisp(c: Connection; o: FS.Server) =
  VAR
    methID := UnmarshalInt(c);
  BEGIN
    IF methID = 0 THEN
      VAR
        n := UnmarshalText(c);
        res: FS.File;
      BEGIN
        res := o.open(n);
        MarshalNetObj(res, c);
        Flush(c.wr)
      END
    ELSE
      < error, non-existent method >
    END
  END
```

The stubs have a narrow interface to the rest of the system: they call the `new` and `free` methods of `Locations` 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`:

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

VAR stubs: Typecode -> StubRec;
```

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

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

(In the actual system the `stubs` table is indexed by stub protocol version as well as type code, to make it easy for a program to support multiple protocol versions.)

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

```
MarshalNetObj(obj: NetObj.T,
              c: Connection) =
  IF obj = NIL THEN
    MarshalInt(-1, c);
    MarshalInt(-1, c)
  ELSE
    IF obj.state = Unexported THEN
      VAR
        i := NewObjID();
      BEGIN
        obj.wr := (SelfID(), i);
        objtbl[(obj.wr.sp, i)] := obj;
        obj.type := Exported;
        obj.disp :=
          Dispatcher(TYPECODE(obj))
      END
    END;
    MarshalInt(obj.wr.sp, c);
    MarshalInt(obj.wr.i, c)
  END

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

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

```
UnmarshalNetObj(c: Connection)
        : NetObj.T =
  VAR
    sp := UnmarshalInt(c);
    i := UnmarshalInt(c);
    wrep := (sp,i);
  BEGIN
    IF sp = -1 THEN
      RETURN NIL
    ELSIF objtbl[wrep] # NIL THEN
      RETURN objtbl[wrep]
    ELSE
      RETURN NewSurrogate(sp, i, c)
    END
  END;
```

The call `NewSurrogate(sp, i, c)` creates a surrogate for the network object whose wire representation is `(sp, i)`, assuming that `c` is a connection to an address space that knows `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:

```
NewSurrogate(sp: SpaceID,
             i: ObjID,
             c: Connection): NetObj.T =
  VAR
    loc := Locate(sp, conn);
    tc := ChooseTypeCode(loc, i);
    res := Allocate(tc);
  BEGIN
    res.wr := (sp, i);
    res.state := Surrogate;
    objtbl[(sp, i)] := res;
    RETURN res
  END
```

The call `Locate(sp, c)` returns a `Location` that generates connections to `sp`, or raises `CallFailed` if this is impossible. It requires that `c` be a connection to an address space that knows about `sp`. The call `ChooseTypeCode(loc, i)` returns the code for the calling address space's surrogate type for the object whose ID is `i` and owner is the address space to which `loc` generates connections. The call `Allocate(tc)` allocates an object with type code `tc`.

To implement `Locate` without resorting to broadcast, each address space maintains information about its own transports and the endpoints of the address spaces it knows about:

```
VAR tr: SEQ[Transport.T];
```

the sequence of transports available in this space, in decreasing order of desirability.

```
VAR names: SpaceID -> SEQ[Endpoint]
```

`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 Locate 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, Locate 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(c, P(args)) to indicate an RPC call to P(args) performed over the connection c. Here is the implementation of Locate:

```
Locate(sp:SpaceID,
       c: Connection): Location =
  IF NOT sp IN domain(names) THEN
    VAR
      backc := c.loc.new();
    BEGIN
      names[sp] :=
        RPC(backc, GetNames(sp));
      c.loc.free(backc)
    END
  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 CallFailed
  END

GetNames(sp) = RETURN names[sp]
```

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 ChooseTypeCode, 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.

```
Dirty(i: ObjID,
      sp: SpaceID): SEQ[Fingerprint] =
  VAR
    tc := TYPE(objtbl[(SelfID(), i)]);
    res: SEQ[FingerPrint] := empty;
  BEGIN
    < Add sp to 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

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

In the above we assume that TCToFP and FPToTC convert between equivalent typecodes and fingerprints and 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 again 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. The marshaling of readers and writers is very similar; 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 creates a back connection `newc := c.loc.new()`, where `c` is the connection over which `rd` is being marshaled, chooses a unique ID, sends the ID over `newc`, sends the ID over `c` as the wire representation of `rd`, and forks a thread that copies data from `rd` into `newc`. In the receiving process, two threads are involved. The thread servicing the connection `newc` 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 `.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 `vsrg`, it calls `vsrg.claim()` and returns the resulting reader.

The claim method of a surrogate voucher `vsrg` begins with `newc := vsrg.loc.new()` and marshals `vsrg` to `newc` (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 `newc` 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 `newc` and the voucher `vsrg` containing the original reader `vsrg.rd`. The dispatcher procedure plays the role of the forked thread in the first design: it reads buffers from `vsrg.rd` and sends them over `newc`.

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. By contrast 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 implemented the second design, since (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 (e.g. 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 paper.

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`, either by ignoring the `B` data fields and methods, or by somehow holding them in reserve. In either case, the new program may operate on the `A` part of the state, either directly or by calling methods of `A`. But there is no reason to imagine 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 accomodate 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, wire it into his application, configure his program to listen at that port and to forge surrogates for the special objects at that port.

The procedure `LocateHost(nd)`, which appeared in our example earlier, simply forges a surrogate for the special object of the agent on the node `nd`, and the procedures `Import` and `Export` use the table in that object.

6 Numbers

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. Some performance numbers are:

```
Null call               3310 usecs/call
Ten integer call        3435 usecs/call
Same object argument    3895 usecs/call
Same object return      4290 usecs/call
New object argument     9148 usecs/call
New object return      10253 usecs/call
Reader test             2824 KBytes/sec
Writer test             2830 KBytes/sec
```

These numbers were taken using Digital workstations equipped with MIPS R3000 processors (25 Specmarks) running Ultrix, equipped with a 100 megabit AN1 network [20]. On this configuration, it takes 1600 usecs for a C program to echo a TCP packet from user space to user space. The remaining 1710 usecs for the null call is spent in two Modula-3 user space context switches (which together add some 235 usecs of latency to the null call) plus the cost of marshaling and unmarshaling the object whose null method is being called. The ten integer call test shows that the incremental cost of an integer argument is about 12 usecs. The first "same object" test shows that the incremental cost of a network object parameter that does not lead to a dirty call is 585 usecs; the second shows that if the object is a return value instead of an argument, the incremental cost is 880 usecs. This extra cost reflects the acknowledgement that must be sent when the result message contains a network object. The "new object" tests determine the overhead of network object arguments and results for which dirty calls have to be made; they are much higher. The last two lines give the thoughput of network streams. For comparison, the thoughput of a raw TCP stream using a C program is about 3400 KBytes/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 paper.)

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.

7 Experience

Our system has been working for a few months. Several projects are building on the system; 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 9000 miles apart). When a new version of a package is checked in at one repository, the siphon will copy it to the other repository within a few hours. Again, all new versions of 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 advantange of the ability to easily plug new transports into the system. 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.

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 [17], 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 [21], [23].

9 Acknowledgements

We are grateful to Bob Ayers, Andrew Black, John Ellis, David Evers, Rivka Ladin, Butler Lampson, Mark Manasse, and Garret Swart for helpful discussions; and to Paul Leach, Sharon Perl, and the referees for helpful comments on the paper.

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 Ted Wobber. Distributed Garbage Collection for Network Objects. Submitted for publication.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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

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

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

[17] 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.

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

[19] 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.

[20] 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. SRC Research report 59, April 1990.

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

[22] 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.

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