Mobile Network Objects

Robert A. Shillner and Edward W. Felten

Dept. of Computer Science
Princeton University
{*ras,felten*}*@cs.princeton.edu*

Abstract

We describe extensions to the DEC SRC Modula-3 network objects system which support migration of network objects between address spaces. Unlike other mobile object systems, we use an existing programming language rather than inventing a new language with object mobility primitives, and we require no changes to the compiler, standard runtime library, or operating system. This paper details the design, implementation and performance of our mobile objects system and two sample applications which use the system.

1 Introduction

The *network objects* paradigm [2] for distributed programming combines object-oriented programming with remote procedure call (RPC). A *network object* is an object whose methods may be invoked by processes other than the one which contains the object. The methods of a network object always execute on the object's home node; if the network object does not reside in the caller's address space, then the system performs a *remote method call*, an action analagous to a remote procedure call.

This paper describes the design, implementation, and performance of a network objects system that supports object mobility. Our implementation is an extension of an existing (non-mobile) network objects package, the DEC SRC Modula-3 network objects system [2]. Our system differs from most mobile-object systems in that we use an existing programming language rather than inventing a new language with object mobility primitives (unlike many other systems [7, 3, 1, 4, 6]), and we require no changes in the compiler, standard runtime library, or operating system (unlike Amber [5]). Despite these constraints, our system maintains the good features of the original DEC SRC network objects system: ease of use, automatic marshalling of complex data structures, reasonable response to failures, and fast execution of local calls. Although our implementation requires the Modula-3 language, our techniques can be used to add mobility to any reasonably functional network objects system.

1.1 Why migrate objects?

It is well known that the power and flexibility of a network objects system are enhanced if the network objects are able to migrate between address spaces. There are many reasons to migrate objects [7]:

- to exploit locality by migrating objects to the sites where they are used most often;
- to balance load between machines by migrating objects from heavily loaded machines to lightly loaded machines;
- to improve scalability by migrating work from a heavily loaded server to its clients;
- to dynamically reconfigure a running distributed program by migrating objects to a newly added server or migrating objects away from a server that is about to shut down.

To illustrate some of the benefits of network object mobility, consider the example of a trivial file service [2]. Figure 1 shows Modula-3 type declarations for a file service network object and a file network object. A client which remotely invokes the file service's `open` method receives a remote handle on a file object. The client can then remotely invoke the file's `getChar` and `eof` methods. If network objects cannot move, then all of the file objects must exist on the server node only. In this case, every `getChar` invocation involves a network round-trip, and the server node becomes a central bottleneck, limiting scalability.

```
TYPE
  FileService = NetObj.T OBJECT
  METHODS
    open(name: TEXT): File;
  END;

  File = NetObj.T OBJECT
  OVERRIDES
    getChar(): CHAR;
    eof(): BOOLEAN;
  END;
```

Figure 1: Object type declarations for a trivial file service.

Network object mobility can improve the system's performance in several ways. For example, a file object can move to the client node that opened it. Subsequent **getChar** operations involve no network communication, and the central server becomes less loaded. The file network object can move back to the server when the client finishes with it.

Another approach involves using multiple servers which migrate network objects between themselves, but which do not pass objects to clients. Under this scheme, a server which holds too many file network objects (or whose file objects are very busy) can migrate some file objects to other servers, thus balancing the workload across all of the servers. The clients remain oblivious to the movement of the file objects, because a remote invocation automatically goes to the current holder of the invoked object. New machines can join the server group by acquiring some of the file objects held by the existing servers, and a machine can leave the group by sending all of its file objects to other servers. The result is a "multi-server:" a multi-node distributed file service that can scale by adding more server nodes.

The following section provides an overview of the original DEC SRC network objects system. Subsequent sections describe the mobility facilities we added, their performance, two sample applications we implemented with the system, and some limitations we see in our approach.

2 Modula-3 network objects

Rather than build a mobile network objects system from scratch, we extended the DEC SRC network objects system [2], a network objects implementation for the Modula-3 programming language [8]. Modula-3 is an object-oriented language, with a strong type system, singly-inherited object types, automatic garbage collection and multiple threads of control. The original DEC SRC network objects system supports stationary network objects. The system fits neatly into Modula-3 programming style, and it leverages the Modula-3 type system to provide powerful marshalling of arbitrary parameter types.

This section reviews the original DEC SRC network objects system. Section 3 describes how we modified this system to support mobility.

2.1 Abstract and concrete network objects

A Modula-3 object is a data record paired with a method suite [8]. Each method has a *signature* (names and types of its arguments plus a return type) and a *body* which contains the method's code. A new object type A' may be declared as a *subtype* of an existing object type A. In this case, type A' *inherits* the data fields and methods of A. A' may declare additional methods and fields, and may optionally *override* the bodies of some of the methods it inherits from A. An important property of subtyping is that a value of type A' may be assigned to a variable of type A (see Figure 2). All Modula-3 objects are reference types: they reside in the garbage-collected heap and are accessed via pointers.

The Modula-3 network objects system uses subtyping and inheritance to separate the interface and implementation of network object methods. The *abstract object type* specifies the public interface to a network object. The abstract type contains the signatures of the methods which may be invoked remotely, but it does not contain any method bodies or data fields; it is a "pure" type. The *concrete object type* implements the network object: it is a subtype of the abstract type, and it implements the abstract type's methods by overriding them. The concrete type contains the data fields which give the network object its state.

No network objects of the abstract type are ever instantiated; instead, the user creates objects of the concrete type. Clients who know only the abstract type can execute a concrete object's methods (since they are overrides of the abstract methods), but they cannot directly access the concrete data fields, which do not exist in the abstract type.

```
TYPE
  A = OBJECT
  METHODS
    foo() := NIL;
  END;

  APrime = A OBJECT
  OVERRIDES
    foo := Foo;
  END;

PROCEDURE Foo(self: APrime) =
  BEGIN
    ...
  END Foo;

VAR
  a: A := NEW(APrime);       (* Legal. *)
  aPrime: APrime := NEW(A); (* Illegal! *)
```

Figure 2: Subtyping and inheritance in Modula-3. Type `A` is an object type with one method, `foo`, the body of which is `NIL`. Type `APrime` is a subtype of `A`. `APrime` inherits `foo` from `A`, but overrides it in order to give it a body, the procedure `Foo`. The definitions of the variables `a` and `aPrime` demonstrate the rules for assignment of subtypes.

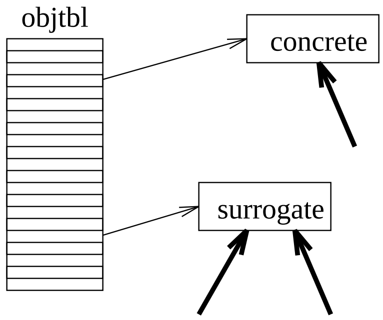

Figure 3: The object table in the original DEC SRC network objects system is a hash table which maps wirereps to concrete and surrogate network objects. The heavy arrows represent the user's pointers to the objects.

2.2 Surrogate objects and remote method calls

When a pointer to a concrete network object is passed across the network, the system creates a *surrogate object* in the recipient. The surrogate acts like a remote handle on the concrete object—methods invoked on the surrogate actually execute at the concrete object. The surrogate's type is a subtype of the abstract type, so it inherits all of the abstract type's methods. The surrogate overrides the abstract methods with stub methods that perform remote procedure calls to the concrete object. As in a traditional RPC system, a stub compiler generates the surrogate type definitions and stub methods from the abstract type.

Arguments and results of remote method calls must be *marshalled* into byte streams for transmission across the network. The Modula-3 network objects system provides a powerful marshalling facility which is general enough to handle almost any data type, including arrays, records, objects, and pointers to arbitrary graph structures. Data structures are marshalled through a process called *pickling*; pickling a data structure involves recursively traversing the structure until all of its state has been encapsulated into a byte stream, or *pickle*. Pickling preserves sharing and pointer cycles. Unmarshalling a pickle creates an exact copy of the original structure. The pickling package is also customizable: the programmer can provide code to specially pickle and unpickle any given datatype.

2.3 Propagation of surrogates

The network objects system uses the custom pickling facility to propagate surrogate objects around the network. When the system pickles a network object (concrete or surrogate), the result is a *wirerep*. A wirerep is a ⟨`spaceID`, `objNum`⟩ pair, where a `spaceID` uniquely identifies an address space, and an `objNum` uniquely identifies concrete objects created in the same address space. The wirerep for a surrogate is equal to the wirerep for the corresponding concrete object. Thus, a wirerep acts as a unique network-wide reference to a particular concrete object.

When a wirerep arrives in an address space, the runtime system must unpickle the wirerep into a pointer to a local object. Each address space maintains an *object table*, which maps wirereps onto pointers to local concrete and surrogate network objects (see Figure 3). If a newly-arrived wirerep has an entry in the object table, then the result of unpickling the wirerep is the network object given in the table entry. If the wirerep has no entry in the object table, then the runtime system must create a surrogate. To do this, the system contacts the concrete object's owner (as given in the `spaceID` field of the wirerep) to discover the type of the concrete object. Then the system creates a new local surrogate object of the appropriate type, puts an entry for the wirerep into the local object table, and returns a pointer to the new surrogate. Contacting a concrete object's owner in this way is called making a *dirty call.*

2.4 Garbage collection and the special object

In addition to providing type information to prospective surrogate-holders, dirty calls also serve to inform the concrete object's owner of the presence of surrogates. The owner records the set of address spaces which hold surrogates for a particular concrete object in that object's *dirty set* — each dirty call adds another address space to the set. As long as a concrete object's dirty set is nonempty, the concrete object must have an entry in its owner's object table, which protects the concrete object from the garbage collector.

When a surrogate-holder's local garbage collector deallocates a surrogate, it makes a *clean call* to the concrete object's owner. A clean call removes its caller from the concrete object's dirty set. When all of the surrogates have called in clean (making the object's dirty set empty), the owner removes the concrete object from its object table. If no local references to the concrete object exist, it may also be garbage collected. Dirty and clean calls provide network-wide reference-counting garbage collection for network objects.

Dirty and clean calls are actually the methods of a distinguished network object called the *special object.* Each address space has one concrete special object, and each address space can construct a surrogate for any other address space's special object. The network objects runtime systems in various address spaces invoke methods of the special objects to communicate with each other. As with all remote method calls, special object method calls may have arguments and return values. For example, the arguments to a dirty call are the wirerep of the object and the address of the node making the call. The return value is the type information from which the caller constructs a surrogate.

3 Object movement

Our goal was to add object migration to the original DEC SRC network objects system without changing the language. The original system has three components: a runtime library, a stub generator and a name-service daemon. We were willing to modify all of these parts, but we did not want to change the Modula-3 compiler, linker, or standard library, or the operating system. These restrictions ruled out features which required thread migration or dynamic code linking. Thus, when an object moves in our system, in-progress remote method calls must execute in their entirety either at the original owner or at the new owner, and any node which wants to acquire ownership of a concrete object must already contain code for the object's methods.

Figure 4 gives a before-and-after illustration of object mobility. The general idea is that a node which holds a surrogate may request ownership of the corresponding concrete object. The object's owner responds to such a request by sending the concrete object to the requestor, who becomes the object's new owner; the old owner becomes an ordinary surrogate-holder. The system eventually informs other surrogate-holders of the object's movement; they create new surrogates, which refer to the object at its new location. In the meantime, the old owner forwards remote method invocations on to the new owner.

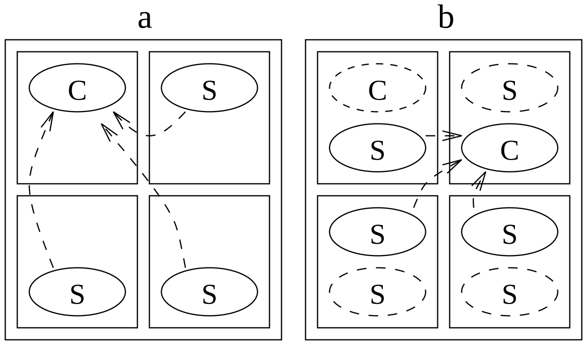

Figure 4: Abstract view of object movement. Figure 4a shows four address spaces. One address space contains a concrete object, and the other three contain surrogates for that object. The dotted arrows show the correspondence between the surrogates and the concrete object. Figure 4b shows the four address spaces after the concrete object has moved to a new owner. The old owner and the other address spaces have new surrogates which refer to the concrete object at its new owner. The old copy of the object and all of the old surrogates are left to be garbage collected.

3.1 Proxy objects

Because our system cannot migrate threads, it cannot move concrete objects which contain active method invocations; objects must be *idle* before they can move. Thus, the system must perform *access monitoring*: it must keep track of how many active calls are in progress at every concrete object in the system. Since concrete objects' methods are written and compiled by the user, we cannot change them, so we must enclose each concrete object in some kind of wrapper that performs access monitoring. We call this wrapper a *proxy*.[1]

Structurally, the proxy is an object with the same abstract type as the concrete object itself. Each proxy contains a pointer to the object it wraps; we say the proxy *guards* the wrapped object. Each method of the proxy contains a call to the corresponding method of the "real" concrete object, sandwiched between two pieces of access monitoring code (as described in section 3.2). Because the system must perform access monitoring for both local and remote calls, user programs must use pointers to the proxy rather than direct pointers to the concrete object. Using proxies is transparent because a proxy exports the same abstract interface as the object it guards, and it forwards all method calls to the guarded object.

Proxies play another important role in object movement. When a concrete object migrates away from an address space, the runtime system replaces it with a surrogate which refers to the concrete object's new remote owner. Since the new surrogate and the old concrete object have different memory addresses, the system must replace all pointers to the old version of the concrete object with pointers to the new surrogate. If user processes could have direct pointers to a mobile object, the runtime system would have to track down all pointers to the object and change them; these pointers could be in the heap, in the stack of any thread, in machine registers, or in a register save area. In our system, the only direct pointer to a local concrete object resides in the object's proxy, and so the runtime system needs to change only the proxy when the object moves.

Our system also guards each surrogate object with a proxy. The system must "repoint" a surrogate object after each movement of the concrete object to which it refers, to ensure that remote method calls made through the surrogate go to the most recent known location of the concrete object. The original Modula-3 network objects system does not contain any facility for "repointing" surrogates. Adding such an operation would involve extensive changes to the runtime library and stub generator to make the "repoint" operation safe for multithreaded programs.

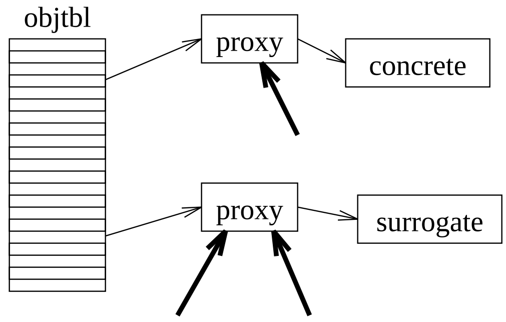

Figure 5: The object table in the mobile objects system maps wirereps to proxy objects. The heavy arrows represent the user's pointers.

Rather than modifying the behavior of surrogate objects in this way, our system guards each surrogate with a proxy object. When the system needs to "repoint" a surrogate, it instead creates a *new* surrogate and updates the proxy so that it guards the new surrogate, leaving the old surrogate to be garbage collected. This strategy allows us to reuse without modification the existing surrogate code from the DEC SRC network objects system.

Guarding surrogates with proxies also provides an easy way to redirect a program's pointers to a formerly remote concrete object which has just migrated to the local node. When a concrete object arrives, the new owner updates its proxy, which previously guarded a surrogate, so that it guards the newly arrived concrete object.

An important modification required by the use of proxies concerns the object table. In the original system (Figure 3), the object table entries are pointers to concrete or surrogate network objects. In the mobile objects system, the table entries instead point to the proxies (Figure 5). This means that the result of unpickling a wirerep is a proxy.

In a client-server application, the client has a proxy which guards a surrogate, and the server has a proxy which guards a concrete object. The client's surrogate refers to the server's proxy. A remote method call executes as follows:

- The client calls a method on its proxy.
- The proxy method calls the corresponding method on the surrogate object which it guards.
- The surrogate method sends the object's wirerep,

[1] We use the term "proxy" in a different sense than does Shapiro [9]. Shapiro's proxies are more like our surrogates.

the method number and the pickled arguments to the network objects runtime system in the server.

- The server's runtime system looks up the wirerep in its object table to obtain a pointer to its proxy object. The runtime then calls the appropriate method of the proxy.

- The proxy calls the corresponding method in the concrete object which it guards.

- The concrete object's method executes.

- The return value goes back to the client by tracing the same steps in reverse.

Note that *two* proxy methods execute, one in the client and one in the server. Although the number of steps required by a remote call is large, the effect of the added steps on performance is small, since the cost of network communication dominates the time for a remote method call.

We modified the existing stub generator to automatically produce the proxy type declarations and method implementations from the abstract network object type. Following is a sample proxy type definition, generated from the abstract type `A` in Figure 2:

```
TYPE
  AProxy = A OBJECT
    OVERRIDES
      foo := ProxyFoo;
  END;

PROCEDURE ProxyFoo(proxy: AProxy) =
  VAR obj: A;
      wasConcrete: BOOLEAN;
  BEGIN
    obj := GetObj(proxy, wasConcrete);
    obj.foo();
    DropObj(proxy, wasConcrete);
  END ProxyMeth;
```

Type `AProxy` is the proxy type, a subtype of `A`. `AProxy` overrides `A`'s `foo` method, setting its body to the procedure `ProxyFoo`.

`ProxyFoo` calls `GetObj` and `DropObj`, two procedures in the mobile objects runtime which perform access monitoring and synchronization. `GetObj` also returns a pointer to the object the proxy guards, which `ProxyFoo` stores in the local variable `obj`. Between the calls to `GetObj` and `DropObj` is the call to `obj.foo`, the corresponding method of the guarded object.

```
PROCEDURE GetObj(proxy: ProxyObj;
                 VAR wasConcrete: BOOLEAN)
                             : AbstractObj =
  VAR
    guardedObj : AbstractObj;
  BEGIN
    LockAcquire(proxy.lock);
    LOOP
      guardedObj := GuardedObject(proxy);
      IF IsSurrogate(guardedObj) THEN
        LockRelease(proxy.lock);
        wasConcrete := FALSE;
        RETURN guardedObj;
      ELSE
        IF proxy.state = Active THEN
          Increment(proxy.refCount);
          LockRelease(proxy.lock);
          wasConcrete := TRUE;
          RETURN guardedObj;
        ELSE
          ConditionWait(proxy.changed);
        END;
      END;
    END;
  END GetObj;

PROCEDURE DropObj(proxy: ProxyObj;
                  wasConcrete: BOOLEAN) =
  BEGIN
    IF wasConcrete THEN
      LockAcquire(proxy.lock);
      Decrement(proxy.refCount);
      IF proxy.refCount = 0 THEN
        ConditionBroadcast(proxy.idle);
      END;
      LockRelease(proxy.lock);
    END;
  END DropObj;
```

Figure 6: Code for `GetObj` and `DropObj`.

3.2 Access Monitoring

Recall that access monitoring keeps track of how many method calls are currently executing on an object; this is necessary in order to detect when objects are idle, since only idle objects can move. Access monitoring is necessary only at proxies which guard concrete objects. It can be omitted at proxies that guard surrogate objects because surrogates never move.

Our access monitoring scheme centers on two properties of a concrete object: its *reference count* and its *state*. The reference count is the number of threads which are executing methods of the object. The state can be one of the following values:

`Active` is the "normal" state of a concrete object. `Active` objects may accept method invocations.

`MovePending` denotes that the object is waiting to be moved. No new method invocations are allowed on `MovePending` objects, but existing method calls continue to run until they return. When they the reference count becomes zero, the object can move.

`InTransit` objects are moving from node to node. No new or existing method invocations are allowed on `InTransit` objects.

`NewlyFetched` objects have just arrived in a new address space, but are not yet ready to receive method invocations. No new method calls are allowed on such an object until its state changes back to `Active`.

Note that only the `Active` state allows new method calls. Thus, while an object is in the `Active` state, the reference count may increase (via method invocation) or decrease (via method return). While an object is in `MovePending` state, the reference count may only decrease, because no new method invocations may begin. The states `InTransit` and `NewlyFetched` require the reference count to be zero; that is, the object must be idle.

`GetObj` and `DropObj` are responsible for maintaining the invariants governing the relationship between a concrete object's state and its reference count. They do this by *stalling* method calls which would violate the invariants. Pseudocode for `GetObj` and `DropObj` appears in Figure 6.

`GetObj` begins by acquiring the lock[2] which guards the access control information for the object. It then gets a pointer to the object guarded by the proxy. If this object is a surrogate, `GetObj` releases the lock and returns immediately.

If the guarded object is a concrete object, `GetObj` checks the state to see whether new method calls are allowed. If so, `GetObj` increments the reference count, releases the lock, and returns. Otherwise, `GetObj` waits on the condition variable `proxy.changed`, which the runtime signals when the object's state changes, or when the proxy is updated to guard a different object. Waiting on `proxy.changed` has the effect of stalling `GetObj` until it is safe to invoke a method.

`DropObj` uses the `wasConcrete` parameter, which is set by `GetObj`, to determine whether `GetObj` incremented the reference count. If so, `DropObj` decrements the reference count. If `DropObj` decrements the reference count to zero, it broadcasts `proxy.idle` in case the runtime system is waiting for the object to become idle in order to move it.

3.3 Mobility API

We chose to make object movement the responsibility of the programmer rather than making objects move automatically. We did this because we wanted to allow programmers to use application-specific object movement policies. We believe it would not be too difficult to create an add-on library that provided automatic mobility for generic objects.

Our movement primitives provide a *pull* interface: the destination address space initiates all object motion. The alternative (*push*) is to allow object owners to initiate transfers. We chose the pull interface for two reasons. First, the pull interface fits naturally with one expected use of the mobility system: a client about to make repeated invocations of an object asks the owner for possession of the object. Second, an object cannot move to an address space that has not linked code for the object's methods; it is easier for the programmer to avoid this pitfall in the pull model.

There are two drawbacks to the pull model. First, if the owner decides it wants to get rid of an object, it cannot do so directly but must ask some other process to fetch the object. Second, the pull model makes it harder for the owner to control the locations to which an object may migrate.

There are five procedures which a programmer uses to control object location and movement. In this section, the "type" `ConcProxy` is a proxy which guards a concrete object, `SrgProxy` is a proxy for a surrogate, and `Conc` is a concrete object. The following two

[2] Actually, in order to make local calls faster, we use an equivalent but more elaborate locking scheme that, for most operations, uses a faster "spin lock" type mechanism rather than a general lock, but falls back on the general lock in some complicated cases. We omit the details in order to simplify the discussion.

routines are called by clients which wish to acquire ownership of mobile objects:

`Fetch(o:SrgProxy):Conc` asks the owner of the concrete object to transfer ownership of the concrete object corresponding to the given surrogate. `Fetch` blocks until the concrete object arrives, then returns a pointer to the object.

`Ready(o:ConcProxy)` makes a newly-fetched concrete object ready to accept remote method calls.

These three procedures are called by owners of concrete objects:

`GetProxy(o:Conc):ConcProxy` returns the proxy for a concrete object. If the object has no proxy, one is created. Any node which creates a new concrete object should use —tt GetProxy to create a proxy for the object, and use only the proxy for further access to the object.

`AwaitFetch(o:ConcProxy)` blocks until some surrogate-holder requests the object, then returns.

`Move(o:ConcProxy)` moves the concrete object to a surrogate-holder that has requested it, then informs all of the other surrogate-holders that the object has moved.

When a server is willing to move a concrete object, it usually forks off a thread running a procedure called the *mobility manager*. This routine is not a part of the mobile network objects runtime, so each application must supply its own mobility manager. This one is typical:

```
PROCEDURE Manager(obj: Conc) =
  VAR proxy := GetProxy(obj);
  BEGIN
    AwaitFetch(proxy);
    <application-specific behavior>
    Move(proxy);
  END Manager;
```

The mobility manager first creates a proxy for the given concrete object using `GetProxy`. Then the thread blocks in `AwaitFetch` until some surrogate-holder requests the object by calling `Fetch`. The mobility manager responds by calling `Move`, but first it gets a chance to perform some application-specific duties. For example, since the pickle package "deeply" copies objects to their destinations, the mobility manager may want to cut pointers from the object to large structures which need not be sent.

If there are any threads executing in the object, `Move` blocks until the object is idle. It does this by waiting on the `proxy.idle` condition variable, which `DropObj` signals when the object becomes idle. After `Move` has sent the object, it acquires a surrogate for the object at its new location, and updates the proxy to guard this surrogate. It then broadcasts the `proxy.changed` condition variable, to awaken any `GetObj` calls which have blocked waiting to make method calls.

A surrogate-holder that wishes to become the owner could use a procedure like this one:

```
PROCEDURE AcquireObject(obj: SrgProxy);
  VAR conc: Conc;
  BEGIN
    conc := Fetch(obj);
    <application-specific behavior>
    Ready(obj);
  END AcquireObject;
```

`AcquireObject` blocks inside `Fetch` until the object arrives. When `Fetch` returns, `AcquireObject` has a chance to execute some application-specific code, perhaps to undo the changes to the object made by the mobility manager just prior to the move. Notice that the parameter `obj` begins as a proxy guarding a surrogate object, but that after `Fetch` returns, `obj` guards the concrete object `conc`.

`AcquireObject` then calls `Ready`, to change the object from the `NewlyFetched` state into the `Active` state, thus reenabling method calls. `Ready` broadcasts the `proxy.changed` condition variable, to awaken any `GetObj` calls which have blocked waiting to make method calls.

3.4 New Special Object Methods

Recall from Section 2.4 that the network objects runtime systems in different address spaces communicate with each other using methods of the special object. We added three new special object methods to support mobility:

Fetch calls are made from a surrogate-holder to the concrete object's current owner when the surrogate-holder wants to acquire ownership of the object.

Migrate calls are made from the concrete object's owner back to a surrogate-holder who had previously made a fetch call, to migrate the object. The migrate call contains the concrete state of the object and carries with it an implicit dirty call from the old owner to the new owner.

NewOwner calls inform surrogate-holders that an object has moved (see section 3.5).

Fetch calls always return immediately, but the object's owner puts each fetch request on a queue (this queue moves with the object). When the owner decides to move the object, it makes a migrate call to the sender of the first fetch request on the queue. The migrate call's implicit dirty call lets the old owner create a surrogate for the newly-moved object without invoking a separate protocol method.

3.5 Locating Objects

When objects can move, it becomes more difficult for potential clients to find them. Since the original network objects system does not have a global object-location service, we chose to use a forwarding-address scheme to track the locations of objects.

When an object moves, its old owner automatically generates a new surrogate, pointing to the object at its new owner. The old owner updates its proxy to guard the newly created surrogate. The result is that remote calls directed to the old owner still work — they go to the old owner's proxy, which calls the old owner's surrogate, which does a remote call to the new owner. Even if the object moves multiple times, calls to an ex-owner are forwarded along the chain of previous owners until they reach the current owner.

If performance and fault tolerance didn't matter, the system as described so far would cope perfectly with object movement. In reality, this solution is unacceptable for two reasons. First, performance of remote calls suffers if they must be forwarded through intermediate nodes before reaching the object's owner. Second, long forwarding chains are undesirable because they can be broken if a node crashes or exits.

The solution to these problems is to make a reasonable effort to keep surrogates up to date — to keep them pointing to the current owner. This constitutes an *eager* approach to disseminating location information. One disadvantage of the eager design is that it can use more messages than a lazier strategy, especially for objects that are not invoked often. In lazier approaches, a little-used object can move several times, producing a long forwarding chain, but without sending any messages to surrogate-holders not directly involved in each move. In contrast, the eager strategy informs every surrogate-holder at every move, resulting in higher network traffic.

When an object moves, our system notifies surrogate-holders by making a *newOwner* call to the special object of each surrogate-holder. These calls are made by an asynchronous thread, since they are not on the critical path. When a node receives *newOwner* information about an object, it creates a new surrogate, pointing to the new owner, and updates its proxy object to guard the newly created surrogate. From this point on, calls go directly to the new owner.

When the new surrogate is built, there might still be remote calls in progress through an old surrogate. There is no harm in this: calls to the old owner go to the old owner's proxy and new surrogate, and then to the new owner. The system does not deallocate an out-of-date surrogate, but leaves it for the garbage collector. Once all calls through an out-of-date surrogate complete, no pointers to it remain, and it becomes subject to garbage collection.

Object mobility also requires that we modify the structure of wirereps. In addition to the existing `spaceID` field, which identifies the original creator of the object, and the existing `objNum` field, which distinguishes the object from all other objects created by the same address space, we add a `locationHint` field. The location hint gives the spaceID of the last known location of the object. When creating a surrogate, the runtime system makes a dirty call to the address space given by the location hint. In most cases, this is the object's current owner. However, if the object has already moved from the address space given by the location hint, that address space accepts the dirty call, but immediately informs the caller of the object's new location via a newOwner call.

3.6 Object Movement Summary

Figure 7 summarizes our design by illustrating object mobility with fetch, migrate and newOwner calls. In Figure 7a, four address spaces have proxies for a concrete object. One of them has the concrete object, while the others have surrogates. Note that the surrogates refer to the concrete object's proxy, not to the concrete object itself.

Figure 7b shows the system after the upper-right node has made a fetch call and the owner has responded with a migrate call. The concrete object now resides in the upper-right node, and that node's proxy now points to the concrete object. Likewise, a surrogate now exists in the old owner, which refers to the proxy in the new owner. The old concrete object and the new owner's surrogate are abandoned and subject to garbage collection. The other two nodes have not yet been informed of the move, and their surrogates continue to refer to the old owner's proxy. Calls made through these out-of-date surrogates work correctly, although they suffer a performance penalty because they must propagate through the old owner.

Figure 7c shows the system after the old owner has made newOwner calls to the lower two address spaces

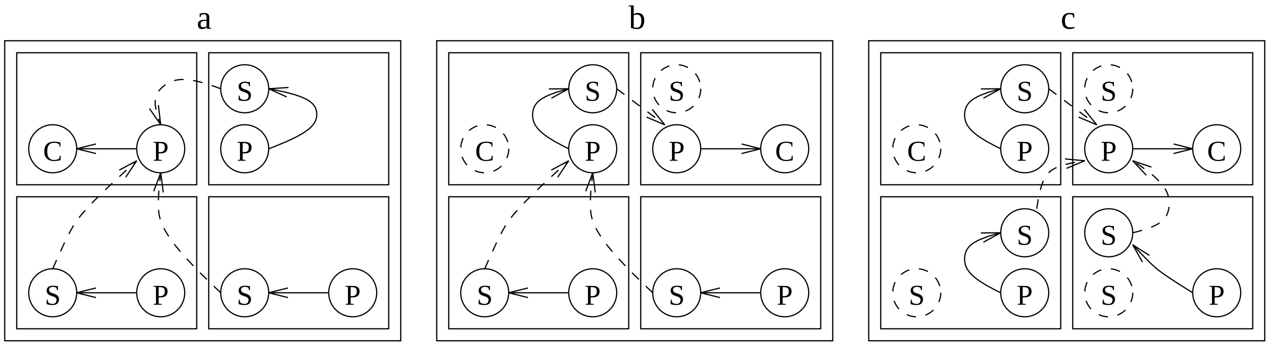

Figure 7: Object movement: (a) start; (b) after fetch and migrate; (c) after newOwner.

Test	Original	Mobile
Local null call	0.128	2.05
Remote null call	1100	1130
Remote ten integer args	1200	1260
Surrogate object arg	1190	1710
Concrete object arg	1230	1730
Concrete object (dirty call)	3090	4660
Move object (0 bytes)		8700
Move object (1k bytes)		16100
Move object (10k bytes)		74500

Figure 8: Performance numbers (in μs) for the mobile objects system. The first column of numbers is for the original DEC SRC network objects system (no object mobility), the second is for our mobile objects system.

and they have replied with dirty calls to the new owner. The dirty calls have resulted in the creation of two new surrogates, which refer to the new owner's proxy. The old surrogates are left to be garbage collected.

4 Performance

We measured the performance of our mobile objects system, testing local and remote method calls as well as object movement. We ran the experiments on a pair of DEC Alpha AXP 3000/500 workstations, running at 150 MHz, and connected by a 10 Mb/s 10Base-T Ethernet. We measured the running time for nine operations: local null method call, remote null method call, remote null method call with ten integer arguments, remote null method call with a surrogate object argument, remote null method call with a concrete object argument, remote null method call with a concrete object argument for which the callee must construct a surrogate, and object migrations of 0 bytes, 1k bytes, and 10k bytes. Figure 8 presents the results.

There is clearly some cost associated with mobility. Local calls to mobile objects incur about two microseconds of overhead. In the original network objects system, users have direct pointers to concrete objects, and calling a local concrete network object's method is no more expensive than a regular method call. In the mobile objects system, users have pointers to proxy objects; the proxy's method must (via `GetObj`) acquire a lock, dereference a pointer (causing runtime type checking), and release the lock before it can call the concrete object's method; another lock and unlock operation occur (in `DropObj`) on the return path. Although this overhead is large compared to an ordinary procedure call, it can be amortized over the amount of work done in a realistic object method.

In both systems, remote calls are much more expensive than local calls. This is not surprising considering the cost of network communication. In both systems, a remote null call requires one network roundtrip, so the two systems have about the same performance. The small mobility overhead is primarily caused by the slightly more complicated data structures required to support mobility.

Calls that pass a mobile object as an argument exhibit higher overhead due to mobility. We are not sure why these overheads are as large as they are, although we do expect there to be some overhead due to an increase in complexity in the object-table implementation in the mobile objects case.

Moving an object requires about nine milliseconds for a null object; larger objects require more time because it takes more time to send their state across the network.

Our experiments imply that performance of some applications may be improved significantly by object migration. Moving a small object across the network is worthwhile if doing so transforms eight or more remote invocations into local ones.

5 Sample applications

We used the mobile network objects system to implement two scalable "client-multiserver" applications: a chat facility and a simple file service. The chat application allows several people (the clients) to monitor one or more *channels*. Anything a user types at a channel is broadcast to everyone monitoring that channel. In a centralized client-server implementation, a central server accepts text from the clients and sends the text to every client monitoring the appropriate channel. This architecture is not scalable: as the number of clients and channels increases, the central server becomes more and more loaded.

Our scalable chat facility uses multiple servers. Each of the servers is in charge of one or more channels. Text sent to a channel goes to that channel's server, which then sends the text to those users monitoring the channel. Since only about twenty people can usefully share the same channel, no single channel can overload a server. The servers migrate the channels among themselves dynamically according to estimated traffic, so the load is always balanced.

Dynamic load balancing makes it easy for servers to join or leave the computation. A server that wants to join adds itself to the list of servers; the load balancing mechanism causes the new server to be given its share of the work. Although we haven't implemented it yet, a server that wants to leave the computation could pretend to be very busy. Load balancing would cause the server's real work to be migrated away; then the server could safely shut itself down.

Our chat application has two important advantages over a single-server approach. First, it scales better as the number of clients becomes very large. Second, it allows servers to be added and removed dynamically without bothering the clients. While it would certainly be possible to implement a similar application "by hand" in the absence of object mobility, our mobility primitives make the programmer's job much easier. If the application was originally built with a single server in mind, mobility — and thus scalability and reconfigurability — could be added as an afterthought.

The second application is a simple distributed file service, similar in many ways to the example given in the introduction. In our file service, multiple nodes acting as servers host file network objects, and clients read and write file data by remotely invoking methods of the file objects. The servers automatically balance the client load among themselves by moving busy file objects to less-loaded servers. One of the server nodes handles all file creates and opens; the load-balancing mechanism distributes the file objects to the other servers as needed.

We ran some experiments with the file service to investigate the performance benefits of the scalable multi-server approach. We measured the rate at which file transfer operations could be performed by eight clients when data was served by a single server, and by three servers. Each client repeatedly wrote and read nine 10-byte files. We warmed up the system by running it for 60 seconds and then recorded the client operation rate over the next 200 seconds. We used DEC Alpha workstations connected by Ethernet, and we noted an 67 percent increase in per-client operation rate, from 99.5 operations/sec to 165.7 operations/sec.

As in the case of the chat application, similar performance gains could have been achieved without using mobile network objects. However, using mobile objects simplified the programmer's job, and led to a cleaner and more extensible design. One possible extension is to allow files to migrate to the clients that use them.

6 Limitations

This section examines some limitations of our system.

6.1 Idle-object movement

Recall that our system only moves objects that are idle. For lightly-loaded objects, this works well; when the system needs to move an object, it does not have to wait long for it to become idle. However, for heavily-used objects, the system may have to wait indefinitely for a moment when no threads are executing the object's methods.

Our current system can deadlock while trying to move concrete objects with certain types of blocking methods. Suppose a thread enters a concrete object's method and blocks waiting for another thread to enter the object (perhaps to release a lock); deadlock will occur if the object owner calls the `Move` procedure on the object. The deadlock occurs because `Move` disallows further method invocations on the object until all threads have exited the object, but the thread blocked in the object's method cannot leave until another thread enters. We could work around this problem by causing `Move` to time out after waiting for some specified period.

A few objects have built-in "daemon" threads, which live for an indefinite time within the object. Because of the idleness requirement, these objects can never move.

6.2 Security

We made no attempt to make our system secure. We feel that network objects system security is a separate topic beyond the scope of this work. Van Doorn *et. al.* [10] discuss one security strategy for the Modula-3 network objects system. We believe our system could be adapted to coexist with their security mechanisms.

6.3 Compatibility

The addition of the location hint to the wirerep makes our system incompatible with the original DEC SRC network objects system. Before old applications (including the nameservice daemon) can work with our mobile objects system, the programmer must insert calls to the `GetProxy` routine, generate new stubs, recompile and relink.

6.4 Fault Tolerance

There is a general tradeoff between fault tolerance and object mobility. In a centralized client-server application, the server represents a single point of failure—if the server goes down, the application stops. However, failure of one or more of the clients does not bring down the entire application. Now consider a mobile network objects application where a centralized server offloads computation to the clients by temporarily moving network objects to clients that are using them. The server still represents a point of failure, but so do the clients—any client that crashes or exits while holding a network object causes that object's state to be permanently lost.

There are several approaches the programmer can take to get around this problem. The server might offload only the objects it can afford to lose. The server could also keep backup copies of the objects, in case the originals are lost.

Another approach is to build "client-multiserver" applications, which use several servers that balance the object load among themselves, without shipping objects to the clients. Since there are fewer servers than clients, and servers are generally less likely to go down than clients, this approach is more robust.

7 Related work

A number of distributed programming systems have used mobility. Emerald [7] and Amber [5] offer object mobility and remote method calls. Mobile objects in both systems have *object descriptors*, which play a role similar to that of a proxy object. Object method invocations first check the descriptor to make sure the object is local; if it is not, the system performs a remote method invocation. Object location information is propagated lazily via a chain of forwarding addresses. Both systems support thread migration for active objects. In addition, Emerald allows code to be moved and dynamically linked to running programs. Amber uses a *global virtual address space*; it lays out memory identically on each node so that the system can locate an object's home node.

The Hermes [3] distributed system also allows objects to move. Hermes uses *temporal address descriptors* (*tads*) to locate objects. A *tad* acts like a wirerep plus a location hint; a *tad* is a statement about the location of an object at some time in the past. Hermes uses a lazy propagation scheme, with a system of progressively more expensive (but more available) techniques to locate objects, looking first locally, then following a chain of forwarding pointers, looking in stable storage, and finally using a highly-available name service. The high availablility of Hermes location techniques works around one of the limitations of the lazy location propagation scheme: its susceptibility to node failures.

Orca [1] replicates objects on many nodes and uses a reliable, totally-ordered broadcast facility to invoke methods on all replicas simulaneously. All object placement is automatic and transparent to the programmer.

Interpreted languages like Obliq [4] and Phantom [6] offer a language-level implementation of network object mobility, rather than the library-level approach we have taken. Interpreted distributed languages have the advantage of easy code propagation across the network. The library-based approach has the advantage of performance.

8 Conclusions

We have described modifications to the DEC SRC Modula-3 network objects system which allow network objects to migrate between nodes. Our system does not require the use of a special language, nor does it require changes in the compiler, standard runtime system, or operating system. Our system retains the simplicity of the original DEC SRC network objects system while providing useful object mobility primitives.

We used our system to develop two scalable applications: a chat facility and a file service. Both applications use mobility to balance load among a number of servers and to incorporate new servers into the running application. These features would have been more difficult to implement using a stationary network objects system.

Topics for future work include how to structure applications to improve fault tolerance, how to support fault-tolerance techniques in the mobile object system, how to integrate strong security mechanisms, and how to automatically move objects around to improve performance, reconfigure services, or balance load.

References

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

[2] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 217–230, 1993.

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

[4] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.

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

[6] Antony Courtney. Phantom: An interpreted language for distributed programming. In *USENIX Conference on Object-Oriented Technologies*, 1995.

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

[8] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[9] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *IEEE International Conference on Distributed Computing Systems*, 1986.

[10] Leendert van Doorn, Martin Abadi, Mike Burrows, and Edward Wobber. Secure network objects. Technical Report IR-385, Vrije Universiteit, Amsterdam, July 1995.