Simple Activation for Distributed Objects

Ann Wollrath, Geoff Wyant, and Jim Waldo
Sun Microsystems Laboratories

Simple Activation for Distributed Objects

Ann Wollrath, Geoff Wyant, and Jim Waldo
Sun Microsystems Laboratories
{ann.wollrath, geoff.wyant, jim.waldo}@east.sun.com

Abstract

In order to support long-lived distributed objects, object activation is required. Activation allows an object to alternate between periods of activity, where the object implementation executes in a process; and periods of dormancy, where the object is on disk and utilizes no system resources.

We describe an activation protocol for distributed object systems. The protocol features overall simplicity as well as applicability to several different activation models. We use the Modula-3 network object system as a base for our implementation; while we make no changes to the underlying network object subsystem, we suggest a minor modification that could be made to the marshalling of network objects to assist in lazy activation, our preferred activation model.

1 Introduction

Distributed object systems are designed to support long-lived persistent objects. Given that these systems will be made up of many thousands (perhaps millions) of such objects, it would be unreasonable for object implementations to become active and remain active, taking up valuable system resources, for indefinite periods of time. In addition, clients need the ability to store persistent references to objects so that communication among objects can be re-established after a system crash, since typically a reference to a distributed object is valid only while the object is active.

Such systems need to provide *activation*, a mechanism for providing persistent references to objects and managing the execution of object implementations. When warranted, object servers can be started up or shut down.

As our platform for distributed objects, we use the network object system [1] provided with the Digital Equipment Corporation System Research Center distribution of the Modula-3 (M3) language [2]. Given that this implementation platform does not support object activation, we have designed an activation protocol to address that need. We identify several important goals for our activation protocol:

- flexible mechanism enabling implementations of a variety of activation models;
- simple activation scheme focused on the core protocol leaving special cases of activation to be specified at a higher layer of abstraction;
- minimal implementation requirements on servers supporting the protocol.

Our implementation of the protocol is constrained by following components of the underlying system:

- the need to be layered on top of the network object system without requiring changes to the network object runtime;
- the protocol should require no modification to existing network object stub generators.

Other factors have influenced the design of the interfaces (more cosmetically):

- interfaces are restricted to single inheritance (although this restriction did not interfere with the design since multiple inheritance was ultimately not required);
- interfaces reflect Modula-3 naming conventions (we have adopted the convention of naming the primary interface in any module **T**, following Modula-3 style definitions).

The activation protocol is specified using the Object Management Group's Interface Definition Language (OMG IDL) [3]. In our protocol definition we use an additional keyword, *serverless*, which denotes a pass-by-value object (i.e., library objects). The serverless feature is not part of standard IDL, but could be thought of (loosely) as an IDL `struct`.

The notion of pass-by-value objects is an attractive one, since exposing a data definition explicitly via `struct` violates data encapsulation. A client possessing a structure may manipulate that structure without restriction, potentially altering the data in a manner not intended or expected by a subsequent recipient. By using a pass-by-value (or serverless) object instead, data can be hidden and a more appropriate interface can control access to and modification of that data. We include serverless objects for just this reason.

The obvious drawback to pass-by-value objects in distributed systems is that the code for the library object implementation must be linked into all clients of such an object. This requirement is much different than having to link in stubs (surrogate code) for a distributed object; all clients must link to the *same* library object implementation. Clients that transmit serverless objects that have a different implementation than the receiver expects will likely cause unanticipated failure: a failure usually encountered during the unmarshalling process performed at the receiver.

2 Activation Models

In a distributed system in which the total number of objects that could be used exceeds the total system resources available, some way of conserving system resource needs to be found. One way of conserving resources is to distinguish between *active* objects, which take up system resources, and *passive* objects, which do not.

More precisely, an *active* object is one that is associated with a process on some system. A *passive* object is one that is not associated with such a process, but which can be brought into an active state. Transforming a passive object into an active object is a process we refer to as *activation*. Activation requires that an object be associated with a process, which may entail loading the code for that object into a process and restoring any persistent state for the object.

In this section, we describe the client's model of activation, followed by a discussion of various activation models and the trade-offs associated with each.

We will refer to the activation models that we will discuss as *eager* (or deep) activation, *lazy* activation, and *split* activation. Eager activation can be characterized as a strategy that maintains as an invariant that each reference within an active object also refers to an active object. Lazy activation, in contrast, is a model that defers activation of an object to the time at which an operation is invoked on that object. Split activation combines aspects of both schemes.

Client Model

Our basic model, from the client side, involves two distinct forms of reference for objects that are (potentially) in a separate address space. The first of these, called the *internal* reference, typically points to a local surrogate or proxy for an object. From the programmatic point of view, such internal references look like references to other, local objects. These internal references are the entities that are manipulated, used for method invocations, and passed around to other objects. In our system, internal references are simple object references that derive from the base class of `NetObj.T`, the Modula-3 Network Object class.

Internal references in this framework, however, are not guaranteed to refer to a valid object over various runs of the code that implements that object. In particular, if an object is made passive (or crashes and is restarted) the internal reference for it may change. If a client wishes to store a reference to a (potentially) remote object as part of the persistent state of that client, some form of reference is needed that will survive those changes. We refer to this form of reference as an *external* reference.

When a client has an external reference to an object, that reference needs to be converted to an internal reference if the client wishes to make use of the object. Such a conversion can require the activation of the referred to object.

Eager Activation

Eager (or deep) activation denotes an activation model whereby an object, and the objects reachable from that object, are activated at once. In this model when a client restores its state, all external object references are presented as internal object references. As part of restoring an internal reference from its external form, the referred-to object is made active. Additionally, the object's implementation restores its state, causing activation of those objects denoted by external references. Thus, when activating a single object, the transitive closure of objects referenced by that specific object is activated.

This model has the advantage that it requires minimal intrusion on the client-side. A small amount of generic code needs to be written that converts object references from their external form to their internal form and to call the appropriate object activation service. This model also has the advantage that it can be determined at the time the object reference is converted to its internal form whether or not that object can be made available.

One primary disadvantage of this model is that it doesn't handle circular chains of reference. Circular chains of reference occur when an object refers back to itself through any number of intermediate references (e.g., A contains a reference to B which contains a reference back to A). An object can also contain a self-reference. In the eager model described above, activating an object containing a reference that eventually refers back to itself (a circular reference) requires activating the object itself and can lead to deadlock unless complicated avoidance mechanisms are employed.

Another problem with this model concerns scalability. In the eager model, an object is activated when its external reference is read from disk rather than when it is first dereferenced. This strategy seriously affects the scalability of activation. Upon reading an object's persistent state, a potentially large number of objects may be activated at once if the object contains many external object references. Activating an entire tree of objects can cause an "activation storm" where cascading activation requests flood the system.

Another disadvantage to this eager approach is that it will activate an object, even if the object is never used by the client. Ideally, unused object should not consume any system resources. Consider a object which contains references to 100 other objects on 100 other machines. When that object restores these references from disk, 100 processes will be created in an eager fashion regardless of whether the references are actually used by the object. Such "storms" seriously effect the overall performance and predictability of the system.

Lazy Activation

Lazy activation defers activating an object until a client's first use (i.e. the first method invocation). This model of activation is typically implemented by generating stubs that check to see if the target object has been made active for this process. We refer to these stubs as *fault blocks*. Each fault block maintains a reference to the target object. If this reference is nil, the target has not been activated for this process. Upon method invocation, the fault block (for that object) then engages in the activation protocol and retains the reference to the newly activated object. From that point on, all fault block stubs forward method invocations to the surrogate object that stands in for the remote object. This scheme is analogous to "object faulting" in persistent object systems [4] or page faulting upon referencing non-resident memory locations.

The lazy activation approach avoids the deadlock problem of eager activation. Except for certain pathological cases, activation of an object never requires the activation of itself even in the case of circular references.

This scheme is also more scalable. Since activation is deferred until the time of first reference, an object is never activated unless explicitly used. Thus, needless process creation (for unused objects) is eliminated as are activation storms.

A major drawback of this approach is that notification that an object is not available will not occur until the first invocation on the object. At this point, the client may have no sensible recovery strategy and its only option is to exit.

Another drawback is that two stubs exist on the client side for each non-local object. The first is the surrogate that performs remote method invocation. The second is the "fault block" which determines the activation status of the object and activates it if needed and forwards invocations to the surrogate.

One must be careful not to expose fault blocks to the client. This exposure can lead to subtle failures since the client is operating under the assumption that it holds a true reference to the object—one that can be passed as an argument or upon which operations can be invoked—not a reference to a fault block.

For the system to function properly, the functionality of fault blocks must be part of the surrogate (or handled somewhere in the runtime). Due to our desire not to modify the network object runtime or stub (surrogate) generators, we chose not to implement lazy activation at this time. Clean integration of lazy activation with the existing system requires modification to marshalling surrogate objects and to the runtime itself.

Split Activation

We chose to use the hybrid model of split activation. The split activation model is one in which the responsibility for activation is divided into two parts. The first part activates a process for the object. The second part activates the object state on the first invocation of that object. Process activation occurs when the object reference is converted from its external form to its internal form. State activation occurs upon first use of the object.

The split model can be implemented by server-side fault blocks that perform an analogous function to the client-side fault blocks used in lazy activation. In this scheme, a server-side fault block checks to see if the target object is active. If not, it activates the object's state (via a server-provided callback). From that point on, the fault block forwards method invocations to the active object. Split activation sits between eager and lazy activation. While this approach does require special handling on the server side, it does not require any changes to the runtime system.

This model avoids the deadlock problem of eager activation. State activation does not occur until first use of the object, thus breaking potential activation cycles. It also avoids the multiple-object problem that the lazy activation approach suffers from. Clients never deal with two forms of the object: the fault block vs. the surrogate. Clients only deal with surrogates.

This scheme also has the advantage that the server can determine at process activation time whether or not the target object can be reached. Thus, clients can learn earlier about the availability of an object than could be done in the lazy activation approach. A client may be in a better state to recover from this situation. Note that, at any point in the future, communication with an object may fail. A client must be able to attempt recovery from this potential occurrence as well. In split activation, while a client of an object has the potential for early knowledge of failure, it does not necessarily mean that the client will be in the *best* state to handle such failure.

There are some disadvantages of this approach. It can be potentially less scalable then lazy activation if servers are not implemented with activation in mind. If the server restores a large set of object references, then a process will be created for each of those objects causing an "activation storm". However, this process will not propagate beyond the first level of a tree. A child won't propagate the activation until its state is needed. Again, this is upon first invocation. Upon first invocation, the child will restore its state. Any non-local object references will cause the next level of the tree/graph to have processes created for objects (though no state will be activated). Thus process activation occurs as a wavefront, but state activation occurs on demand.

The split activation model transfers control of when activation occurs from the runtime system to the server. A server can choose which objects are activated by restoring its persistent state selectively. We have designed a generic container class to deal with such selective activation of objects. Operations that access elements of the container handle the machinery of activation, thus emulating lazy activation for those objects in the container. Using the container abstraction allows the server to delegate the manual control of activation.

3 The Basic Activation Protocol

This section describes the basic activation protocol in detail. The protocol involves three entities: a client, an activator, and a server process/object. The activation protocol proceeds as follows (starting with the client):

1. *Obtain an external object reference.* Let's say a client wishes to retain a reference to an object for some period of time and to be able to store that reference on disk. From a name service or the object itself (using an internal reference to the object), the client obtains an external reference for that object. The abstraction for external object references will be discussed in the next subsection.

2. *Request activation.* At a later time, this external object reference (obtained previously) can be converted into an internal reference using an *activator*, on the same host as the referenced object, as facilitator. An activator process (daemon) runs on each host in the system. It is the responsibility of the activator to spawn servers (on the local host) corresponding to certain external references. The client hands the activator an external reference together with a request to "activate" that object.

3. *Spawn server process*. Upon receiving the client's request for activation, the activator spawns a server process for the object, passing it relevant data (from the external reference) for bootstrapping purposes.

4. *Server activates*. The server process starts up and sends the internal reference for the object back to the activator, thus informing the activator that the object's activation is complete.

5. *Reply with internal reference to client*. The activator replies to the client with the internal reference that it received from the server process. The client is free to invoke operations on the activated object, using its internal (programmatic) reference.

The client mentioned in the above protocol does not necessarily denote the client application program. An ideal implementation would completely hide the mechanisms of activation from the client.

Activation Interfaces

The interfaces that embody the activation protocol described above are contained in the module Activation defined in IDL (see Figure 1).

```
#include "VantageID.idl"
#include "VantageObj.idl"

module Activation {

   exception UnableToActivate {};
   exception AlreadyActive {};
   exception UnableToDeactivate {};
   exception AlreadyManaged {};
   exception WrongActivator {};

   typedef VantageID::T ID_T;

   interface Address_T : serverless {
      string toText();
      boolean equal(in Address_T addr);
   };

   interface Token_T : serverless {
      VantageID::T vid ();
      string executableFileName ();
      string activationData ();
      Address_T activatorAddress ();
      boolean isManaged ();
      string toText ();
   };

   interface Activatable_T :
                  VantageObj::T {
      Token_T activationToken ();
   };

   interface Manager_T {
      Activatable_T activate (
            in Token_T token,
            in ID_T activationID);
   };

   interface Activator_T {
      Address_T address ();
      Activatable_T activate (
            in Token_T token)
         raises (UnableToActivate,
                  WrongActivator);
      void managed (
            in Manager_T manager,
            in string executableFileName)
         raises (AlreadyManaged);
      ID_T activated (
            in Activatable_T activatedObj,
            in Token_T token)
         raises (AlreadyActive,
                  WrongActivator);
      boolean isActive(
            in Token_T token)
         raises (WrongActivator);
      void deactivated (
            in ID_T id,
            in Token_T token)
         raises (UnableToDeactivate,
                  WrongActivator);
   };
};
```

Figure 1. Module Activation

The Activation module refers to two other interface definition files. The interface described in the file “VantageObj.idl” is one supported by all objects in the overall system we are constructing. This interface, VantageObj::T, contains a single method, which returns an identifier that, with a high degree of probability, uniquely identifies the object. The interface VantageID::T, contained in the file “VantageID.idl”, defines these unique identifiers for objects.

The major interfaces in the Activation module are:

- Address_T - abstraction for object location (pass-by-value),
- Token_T - abstraction for external object references (also pass-by-value),
- Activatable_T - interface supported by those objects capable of being activated,
- Activator_T - interface to the activator,
- Manager_T - interface for object servers that wish to support multiple objects per server process.

The following subsections will describe each of these interfaces in turn.

External Object References

An external reference to an object must contain enough information so that some mechanism can initiate the object’s execution. That is, given some form of external object reference, a client can obtain an internal reference to an active object.

The minimum information needed to initiate a server process is the location of the server program (i.e., executable) and the physical host location for the spawned server process. Additionally, the server may need some bootstrap information in order to read its persistent state, or export itself to a name service, for example.

In our system, external references are known as *activation tokens* (represented by the Token_T interface). An activation token (or simply a token) can be thought of as the meta-data for an object. These tokens must supply the following information:

- a unique identifier for an object,
- the *address* of the object's activator,
- the executable file name of the server program,
- server-specified activation data, and
- a flag indicating whether this object is *managed.*

Each object in our system exports a unique identifier (also referred to as a *vantage ID* or *VID*) which is used for object identification purposes by both the activator and the spawned server itself. We use these probabilistically unique identifiers in order to determine object equality, since reference equality for distributed objects is not supported in most systems. The unique identifier in the activation token, obtained via the vid operation, corresponds to the object's unique identifier.

The address of the activator, obtained as a result of the activatorAddress operation, is the abstract "location" of the activator and is implementation specific. The abstraction for an address is the Address_T interface which contains two operations toText (for converting the abstract representation to string form) and equal (for establishing equivalence of addresses). In our system, an activator address essentially refers to the activator's host.

Note that both Address_T and Token_T types are denoted by the serverless keyword. Both types are represented as pass-by-value objects as opposed to full-fledged distributed objects which would carry the overhead of remote method invocation for invoking each of their operations. Addresses and activation tokens must be light weight entities. Many implementation difficulties would arise if these objects were to be distributed in nature (that is, pass-by-reference).

The executable file name of the server program is the pathname to the server executable (returned via the executableFileName operation). The activator uses this information in the activation token to spawn a server for the object.

The activationData contained in the activation token is entirely under the control of the object (server) for which the token is created. It is up to the object implementation to decide what information is relevant to start-up, and place that information in the token. For example, the activation data for a server might be a pathname to a log file for recovery.

The isManaged flag simply indicates whether an object should share the same server process as other objects on the same node which share the same executable file name. In a later section, we explain our scheme for managing multiple objects within a single server.

In our system, activation tokens are self-contained entities. That is, the activator needs only the information present in a token to spawn a server process. An alternative design (and potentially more flexible) would be for an activation token to refer implicitly to a database containing activation information for objects. For example, a unique identifier would be the only required component of an activation token if the client consulted a database to determine the activator for an object, and each activator consulted a database for further information on the activation attributes of an object (such as its executable file name).

While this design may be more flexible—since it does not fix the contents of activation tokens—it does require that each activator have access to state information, i.e., the database of activation information. This introduces more complexity into the activator and additional failure modes during activation if databases become temporarily inaccessible. Management of such databases (registration of information, update, and querying) also increases the overhead of the activation mechanism.

Note that the address of the activator is embedded in the activation token. Since activation tokens can be stored persistently, an activation token for an object must not change over time, unless all objects to which the activation token was given can be contacted with the value of the new token for the object. Thus, object mobility cannot be easily employed since the contents of activation tokens are fixed at creation time. Other mechanisms must be built to handle moving objects in the system. Important services that are likely to relocate in the future may be stored in a name server and referenced by name rather than strictly by token.

Server Requirements

For an object to be capable of being activated, it must support the Activatable_T interface. Such an "activatable" object need only implement one operation, activationToken, that constructs an activation token containing the appropriate meta-data for the object. As described above, this token contains the object's identifier, executable file name, bootstrap data (activation data), and the address of the object's local activator; an activatable object can query the local activator (which should be a well-known entity) for its address for inclusion in the activation token.

It is the responsibility of each activatable object to hand out activation tokens to those objects requesting the information. Also, an activatable object is responsible for informing the activator when it has completed activation.

The simplest implementation of a server is a single object per server process. In our system, we have the additional notion of *aggregate* objects. An *aggregate* object has an object identity spanning multiple objects (in the same address space), but which functions as a single cohesive unit. Objects in the aggregate share the same identity and cooperate to implement the object's interface. All objects which share the same unique identifier (VID) are considered part of the same aggregate object. If the aggregate for a particular object identifier is "activatable", then all objects which make up the aggregate must implement the Activation::Activatable_T interface.

A server started up by the activator is passed several arguments:

```
-vid <ID> -activationData <data>
```

When the server starts up, it parses the arguments, activates itself, and informs the activator that it is activated. In the process of "activating itself" the server must be careful not to block. In an implementation of a lazy model of activation, blocking might be less of a problem. However, blocking still is a potential hazard if the object attempts to contact any other objects (for example during the server recovery process) before replying "activated" to the activator. Therefore, the server must invoke the activated operation on the activator as soon as possible, and before blocking, so that it will not hold up the client requesting activation or cause potential deadlock.

For aggregate objects, all objects within the aggregate can be activated at once by invoking the activated method on the activator for each activatable object in the aggregate (each having different activation data).

To avoid potential race conditions between servers starting up by hand, or being started via the activator, a server that receives the exceptional return AlreadyActive from the activated operation should exit immediately. This exception means that two servers with the same identity were either started in parallel or a server was started mistakenly by some means. In either case, it is a fatal error upon start-up.

Any activatable object that is created by a server started by hand or is created within a currently running server must register with the activator (via the activated method) before handing out its reference to any client. If this convention is not followed, the activator will not know that such an object is activated, and may start up a duplicate server with the same identity sometime in the future.

The Activator

In our protocol as defined, an activator need not maintain any persistent state, and therefore requires no recovery mechanism. There are two guarantees that the activator must make for the system to function properly:

- like all system daemons, the activator must remain running while the machine is up, and
- the activator must not start servers for those objects which the activator has been informed are active.

The activator maintains a few tables, in memory, of objects that it has participated in activating (or has been informed are activated). These tables are kept in order to enforce the latter guarantee.

The activator supports several operations: address, activate, managed, activated, isActive and deactivated. The managed operation is used in the model where there are multiple objects per server process (or shared server model). We defer discussion of managed objects to the next section.

The address operation returns the abstract location of the activator. As mentioned earlier, the representation of an address is implementation specific.

The activate and activated operations make up the core of the activation protocol. To obtain the internal object reference corresponding to some external reference, a client invokes the activate operation on the activator, located at the address represented in the token, and passes the activation token as the argument. This operation may or may not initiate server execution depending on the status of the object (active or passive).

If the object is not already active, the activator uses the information present in token to spawn a process for the server, passing the server its identifier and activation data as arguments. If activation fails for some reason (failure to spawn server due to lack of system resources, for example), the activator raises the UnableToActivate exception to the client.

When server-side activation is complete, the server process informs the activator by invoking the activated operation on its local activator passing the object's (internal) reference as the argument. The server receives an activation identifier (of type ID_T) as a result of the operation which denotes this "instance" of activation for the object. This identifier is used in *deactivating* the object (explained below). In servicing future activation requests for the same object (with the same activation token), the activator replies with the internal reference previously reported by the server. That internal reference is valid until the object (server) deactivates.

The activator, upon receiving notification of the object's active status from the server, replies to the client with the internal reference for the active object. This internal reference is of type Activatable_T, since all activatable objects support the Activation::Activatable_T interface. The client can then narrow this reference (using a type-safe narrow), to obtain an object of the correct type on which it can invoke operations. Figure 2 illustrates the core activation protocol.

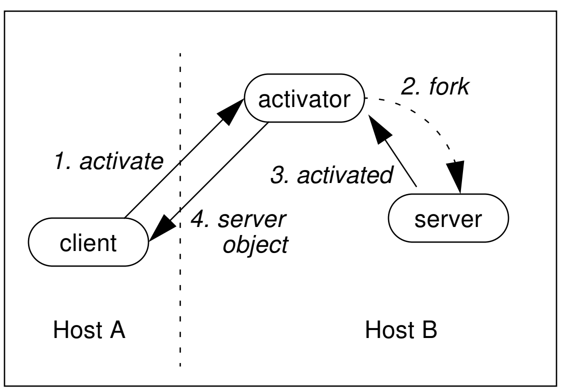

Figure 2. Core Activation Protocol

A server object that has not serviced any requests for some (lengthy) period of time, may wish to shut down or *deactivate* in order to free up system resources. In order to deactivate an object, the server calls the deactivated operation passing two parameters: the activation identifier id (returned by the activated call) and the token of the activatable object. Once an object is inactive, the activator can remove all knowledge of the object from its tables. The activation identifier returned by the activated call and supplied to the deactivated call is employed so that late or duplicate notifications of deactivation will not cause the object to be erroneously forgotten by the activator. The activator uses the identifier to distinguish between valid *instances* of the activation protocol for a particular object.

Note that it is possible for a server to crash without invoking the deactivated operation. This means that the activator will have a stale (or orphaned) internal reference for the object. Thus, clients will receive a stale object reference for the object until the activator detects the object's demise and flushes the stale reference from its tables. The network object system has a notifier facility for detecting object failure. In the activator implementation, we use this facility to detect orphaned references and clean up accordingly. If the underlying system does not provide such a facility, the activator must itself monitor server process status to clean up orphaned references.

A client can check the status of an object by invoking the isActive operation passing the activation token as an argument. This operation returns TRUE if the object corresponding to the activation token is currently executing in a process and returns FALSE if the object is currently dormant (inactive).

In each of the above mentioned activator operations, the exception WrongActivator is raised if the client directs a request to an inappropriate activator. This situation can happen due to a programmer error. Therefore, if the token parameter does not contain the same address as the one for the destination activator, then the request is not carried out and the exception is raised by that activator.

Managed Objects

Up to this point, the activation protocol described enables one active object (with the same unique identifier) per process. If the server programmer wishes to have multiple objects per process, a private protocol would have to be employed between active servers so that subsequent objects can be activated in the same process as the first one. The situation of activating a group of objects in a process is a common one, so we have enhanced the protocol, via the Manager_T interface, to handle this frequently occurring case.

A few simple additions to the basic activation protocol accomplish activation of a group of objects in a single process (the group of objects being those that share the same executable file name). Objects using this shared server model set the isManaged flag in the activation token according to whether the object is "managed"— that is, it should be activated in the same process as other objects on the same node which share the same executable file name.

We introduce the notion of a manager of objects which is responsible for handling activation of a set of objects in a single server process. The Manager_T interface consists of one operation which embodies its activation management function.

When the activator sees an activation token indicating a managed object, it creates a process (with the switch "-exec <executableFileName>" just in case the manager can't figure this out on its own) to manage objects instead of creating one process to manage the one object. Once the manager process (executableFileName) starts up, it makes a call back to the activator, using the managed operation, passing both a reference to itself and the name of the executable for those objects which it manages.

This callback informs the activator that the manager will manage the activation of all objects with the specified executable. The activator can then forward the original activation request to the manager via the manager's activate operation. Since this call is synchronous, the manager does not need to make the activated callback to the activator; when the call returns, activation is complete and the activator can reply to the client.

Tokens subsequently received by the activator which are "managed" and have the same executable file name as a current manager will have their activation request forwarded to that manager directly via its activate operation.

The activator will still handle activate requests in the usual way if the isManaged flag is FALSE.

We could take the radical approach that all objects are managed. This approach would allow elimination of the isManaged flag in the token and the activated operation in the activator.

4 Implementation and Results

The initial implementation of the activator (a network object supporting the Activation::Activator interface) is 635 lines of Modula-3 code (including comments and excluding generic interface and module expansions). The first implementation supports concurrent activation/deactivation requests, but does not include any manager functionality (a small addition to the activator).

Performance tests were run on a dual processor SPARCstation 20 with 128 MB of memory. The first series of tests consisted of a client, the activator, and a server on the same machine.

A server object that is dormant needs to be fully activated by the activator; that is the activator needs to create a process for the server and obtain the internal reference for the object. Such *full* activation takes 700 milliseconds for client, activator and server running on the same machine, all in separate address spaces. This timing includes round-trip latency between client and activator, and process creation time. Most of the overhead associated with activation is spawning a new process for the object server which takes roughly 550 milliseconds. Additionally, activation time is effected by the amount of time the server takes to initialize itself before replying to the activator that the server has completed activation (by invoking the operation activated). We keep the server initialization time to a minimum, only performing the following steps in the server: parse arguments, import the activator, create a server object, and reply immediately to the activator. Such server-side initialization takes about 100 milliseconds. Table 1 shows the breakdown for full activation on a single machine.

If an object server is currently running (that is previously activated via the activator), the activation procedure consists of the activator handing back to the client an in-memory reference for the server object. For client, activator, and object server on the same machine, this scenario takes 3.6 milliseconds and mostly reflects the time for a remote method invocation between client and activator.

When the client and activator are on different machines, full activation takes 692 milliseconds. For this same configuration, handing to the client an already activated object takes 4.6 milliseconds. It is interesting to note that full activation for a remote server takes less time than activation where both client and server reside on a single machine. The increased latency for the local case is due to contention for system resources between client and activator. Table 1 (below) summarizes our performance results (in milliseconds).

Activation performance:	
full activation (local)	700
full activation (remote)	692
simple activation (local)	3.6
simple activation (remote)	4.6

Breakdown for full activation:	
fork server	550
server execution	100
other activation overhead	40
(including: GC ~4 ms; and communication with client ~3-5 ms)	
Other baseline measurements:	
fork trivial Modula-3 program	151
fork empty Perl script	71
fork trivial C++ program	61
fork empty Tcl script	52
fork trivial C program	24

Table 1. Activation Results (in milliseconds)

5 Related Work

The Common Object Request Broker Architecture (CORBA [3]) outlines a protocol for activation. This protocol, specified as part of the Basic Object Adaptor (BOA), attempts to be a completely general solution for activation providing sophisticated activation dependency features such as co-activation and group activation. While these features may be needed in some applications, our experience has shown that a simpler protocol suffices for most applications; we have chosen to focus on a clean protocol rather than an ideal API.

The CORBA activation protocol embodies a multitude of activation features which puts an unnecessary burden upon both server implementations and implementations of the activation mechanism itself, if such extended features are not required by server applications. We use a simpler model that supports activation of a single object within a server, or multiple "managed" objects within a server process. If more complex functionality is required, this functionality can be layered upon our activation protocol.

6 Discussion

Object mobility. Currently, an activation token for an object can not change over time (tokens are fixed when created). Since the location of the server machine is embedded in the token (implicitly in the activator address), objects can not be moved to another machine once a token for that object has been handed out. To overcome the restriction on mobility, a level of indirection could be built into the token, but this could have a significant impact on the performance and reliability of the activation process.

As an alternative form of indirect reference, names could be used as external references for long-lived objects. Thus given a name, a client can contact a name service to obtain an internal reference to an active object currently associated with that name.

Another scheme that supports object mobility is one in which a relocated object leaves, at its old location, a *forwarding pointer* (or "tombstone") [5] that indicates the new location for an object. Using forwarding information left for an object, an activator receiving activation requests at an object's old location can redirect such requests to its new location for a period of time.

The problems with this approach are three-fold. First, the indirection introduced by employing forwarding pointers can increase the overhead of activation, since each activator must deal with both servicing and forwarding requests; this dual-role can cause a potential bottleneck at the activator. Additionally, multiple redirections may occur for requests involving frequently relocated objects, thus increasing the overall service time for activation requests. Finally, forwarding information must eventually be removed, but determining when it is "safe" to remove such information is difficult; there is no easy way to determine how many objects, and specifically which objects, contain references to a relocated object's previous address.

Server implementation requirements. Activation is not a passive protocol. It requires participation and cooperation on the part of the server. Ill-behaved servers can potentially cause deadlock during activation of that server. Servers that fail to inform the activator when they become active can cause unpredictable occurrences such as duplicate servers executing with the same identity. Since we allow servers to be started without activator assistance, informing the activator of an object's status becomes even more vital.

Type system. In our system, external references can be exposed to both clients and servers alike. This additional form of reference is outside the type system and therefore can not be checked statically. In large scale systems, non-type-checked references pose many implementation hazards. If external references were completely hidden from the programmer, this would not be as much of an issue. Since no real type information is present in a token, the client and server programs must be careful to perform a typesafe narrow on a reference resulting from an activation request. An activation strategy that utilized activation tokens that are typed to reflect the objects they represent would overcome this deficiency. Thus ideally, either external references need to be integrated with the type system or the runtime system must hide this form of reference from the programmer.

Need more help from the runtime. The lesson here is that you really do need help from the runtime to make activation seamless. Lazy activation is preferable, and is not easily layered on an already existing system (such as network objects).

As an optimization, the runtime could transmit along with the object its activation token, so that it is more easily accessible to a remote process, and a remote method invocation is not required to obtain the token. This saves times as well as eliminates a potential failure mode if the object from which the token is being obtained could not be reached for some reason (e.g., network failure or processor crash). This scheme would require modifications to the runtime to be completely transparent to the client. Of course, the activation token could be transmitted as a parameter in all interfaces, but this exposes activation at the wrong level.

The problem with making the decision not to tweak the runtime is that artifacts of our activation protocol end up creeping into other interfaces that we design (as with UUIDs needing to be transmitted along with a network object references). If we modified the runtime, object identifiers could be transmitted along with object references. If these two were transmitted as one unit, then testing for equality would be little overhead and not require a remote method invocation to obtain the identifier for comparison. Also, we would not have to expose object identifiers explicitly in our interfaces since objects would inherently have a determined, easily accessible identity.

7 Availability

The Modula-3 source code for the activator and other libraries is freely available. In order to obtain the release, please contact the authors for more information.

References

[1] Birrell, Andrew, Greg Nelson, Susan Owicki, and Edward Wobber, "Network Objects," Digital Equipment Corporation Systems Research Center Technical Report 115 (1994).

[2] Nelson, Greg (ed.), *Systems Programming with Modula-3*, Prentice Hall (1991).

[3] The Object Management Group. "Common Object Request Broker: Architecture and Specification." *OMG Document Number 91.12.1* (1991).

[4] Hosking, Antony L., and J. Eliot B. Moss, "Towards Compile-Time Optimisations for Persistence." In *Implementing Persistent Object Bases: Principles and Practice—The Fourth Int'l Workshop in Persistent Object Systems*, Morgan Kaufmann Publishers, Inc. (1990), pp. 17-27.

[5] Jul, Eric, Henry Levy, Norman Hutchinson, and Andrew Black, "Fine-Grained Mobility in the Emerald System." In *ACM Transactions on Computer Systems 6*, 1 (February 1988), pp. 109-133.