Object replacement using dynamic proxy updates

To cite this article: S Menon and R J LeBlanc Jr 1994 Distrib. Syst. Engng. 1 271

View the article online for updates and enhancements.

Related content
- A review of four distribution infrastructures
  J O'Connell, N Edwards and R Cole
- DC++: distributed object-oriented system support on top of OSF DCE
  A B Schill and M U Mock
- An overview of real-time ANSAware 1.0
  Guangxing Li
Object replacement using dynamic proxy updates

S Menon†‡§ and R J LeBlanc Jr†‡

College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA

Abstract. In a distributed environment, a client program bound to a server fails when the server changes (possibly due to the server being relocated, replicated or reconfigured). In this paper, we describe the design of an object-replacement scheme in a client-server environment. Our design addresses the problem of replacing a server and transparently updating the client handle so that no service interruption is experienced by the client.

The programming environment, on which this work is based, provides passive shared objects, threads, and invocations as the building blocks for distributed applications. Our design takes into account the potential concurrency within clients and servers. Our task was relatively simplified by building the replacement mechanisms on top of an asynchronous event notification facility that handles events on a per-application basis, as opposed to a per-task or per-thread basis.

1. Introduction

Distributed applications are typically structured as a collection of programs (or objects) communicating using messages or remote procedure calls [19]. Most of the existing distributed applications are built using the active object paradigm, i.e. the server objects always have a process associated with them [7,15]. Due to the inordinate amount of system state (stack variables, register values, process information, etc) associated with the server, the active object paradigm makes it difficult to perform maintenance operations such as reconfiguring a server, migrating a server to a new node (for load balancing or maintenance operations on a node) [12].

In the active-object model, a process or a thread is bound to an object, on a machine. Threads are 'light-weight' entities and exist only with an object's address space. A distributed application is composed of a set of active objects spanning the network. Processes communicate by sending messages to each other or by issuing remote procedure calls. Even when using remote procedure calls from a client to a server, no direct state sharing (process state), between the client and the server is performed. This is in contrast to the spirit of a local procedure call, where the process state of the global computation is visible from the caller as well as the called procedure.

Passive objects, in contrast, exist without any process bound to them. When a computation is started, a thread is created and dispatched to the entry point within an object.

* This work was supported in part by NSF contract CCR-86-19886.
† E-mail address: srm@cc.gatech.edu
‡ E-mail address: nick@cc.gatech.edu
§ Present address: Silicon Graphics Inc, Mountainview, CA 94043, USA.
(E-mail address: srm@sgi.com)

Invocations between objects are carried out identical in form to a purely local procedure call; the current thread is dispatched to the new object and the thread's state is visible uniformly.

As a consequence, in addition to the parameters required for the invocation, certain critical state information, such as the I/O channels associated with the client thread and signals caught by the client, is visible in the server. It can be argued that since there is no process associated with the server, it should be relatively easier to reconfigure the service.

The Clouds distributed operating system and the associated programming environment were developed to gather experiences with programming a passive object-based system.

Our experiences with programming the system has been very favourable and seems to reaffirm our assertion about the superiority of the passive-object based system in developing distributed applications. Passive servers are easier to reconfigure, migrate, replicate and make persistent [11,3,18]. We believe that the lessons we learned have wider applicability and key ideas could be adapted to the more common process-based environments provided by platforms such as UNIX.

In this paper, we describe the design of an object replacement scheme in a client-server environment. Our design addresses the problem of replacing a server (possibly due to the server being relocated, replicated or reconfigured) and transparently updating the client handle (server information cached by the client) so that no service interruption is experienced by the client. The scheme we propose is an improvement over existing name binding schemes such as the Xerox Clearinghouse mechanism, where the name binding takes place statically and dynamic name (instance) replacements are not allowed [21].
The object replacement scheme is built on top of a general purpose asynchronous event notification facility [17]. The event primitives are designed to handle notifications in a distributed programming environment where servers are passive, concurrently shared, and distributed across the network. Moreover, the event handling scheme is designed to handle events on a per-application basis (as opposed to the per-thread or per-task basis found in designs such as the Mach exception handling facility). This makes it possible to perform the replacement operations transparently, i.e., without any client intervention.

The asynchronous event notification facility was originally designed to support language features such as exception handling (resumption and termination models) and standard tools such as debuggers and monitors for distributed applications. However, we soon realized that the expressive power of events (compared to simple messages) along with the ability to run a handler using any desired thread-of-control (sender of the event’s or receiver of the event’s) makes it easy to implement a variety of distributed administration tools such as reconfiguration managers, smarter name servers, etc [16].

The rest of this paper is organized as follows: in section 2, we briefly discuss the shared passive-object programming model, followed by the features of the asynchronous event notification facility. In section 3, we outline the related work in supporting server replacements and migrations, discuss their limitations and outline our design goals. In section 4, we discuss the use of proxies in object invocations and how proxies use event handlers to find a new server object, when a server that was bound to a client is no longer available. Supporting object replacements due to reconfigurations requires the name server to understand the inheritance relationship among objects. In section 5, we outline the design of name server that allows clients to be bound to new derivations of a server resulting from reconfigurations, and discuss how the proxy update scheme handles server migrations, replacements, and replicas introduced for load balancing. Finally, in section 6, we discuss our plans for adapting the scheme to active-object based models of client-server computing, the performance impact due to the dynamic update scheme and provide concluding remarks.

2. Shared objects and computation threads

The asynchronous event notification facility was designed for a distributed environment that supports programming with concurrently shared (even by unrelated applications) objects [17]. We view a distributed application as a collection of objects, distributed across the network and linked together by using heavy-weight threads. Objects are large-grained—i.e., at the granularity of protected address spaces. In the shared passive-object model, threads migrate from object-to-object during object invocations. Thus, unlike their counterparts in active-object systems, these threads span address spaces (and machine boundaries). Hence any state attributed to the thread is visible uniformly to all objects the threads visit, regardless of which node each object is bound to. To deal with concurrency within an object, any application-specific information is associated with the thread executing on behalf of the application. Since the threads carry attributes specific to each computation (or application), we use the term computation-threads to distinguish them from the more common usage of the term ‘threads’ (which are light-weight and do not migrate across address spaces).

Thus the view presented to applications is one of mapping various objects into a thread’s address space. This is shown in figure 2.

Based on this view, the programming environment supports the following:

(i) Object invocations (local and remote) that mimic single address space procedure calls. With the view presented above, data required by the invocation as well as the state of the computation (such as I/O channels, signaling information, consistency information [5]) is visible to the invoked object.

(ii) Concurrent sharing by unrelated applications. Since the thread-specific information (such as the executing thread’s stack) is not a part of the object’s address space, unrelated threads may access an object concurrently. Of course, the concurrent access is subject to synchronization constrains, dictated by the object programmer.

2.1. Shared objects and events

We designed the asynchronous event notification facility to take into account item (ii) described above (concurrent access by unrelated applications) and to exploit the system view described in item (i). Thread-based event notification localizes the effect of an event to the application for which it is intended. To implement application-wide event notification using threads, a thread is attributed with the requisite information. As mentioned earlier, object-invocation in the passive object paradigm is carried out by using the invoking object’s thread of control (logically). Thus attaching event handling information to a thread supports the following two features:

- Since the same logical thread is used to invoke all the objects partaking in the computation, the attached event
information is available for examination at all nodes. This makes it possible for event handling to be "seamless," across object as well as node boundaries.

- Since events are attached to a thread, it makes it possible for multi-threaded servers to be handling clients with different event handling requirements.

In short, attaching necessary state information to the thread solves both the problems of distribution and concurrency.

In our design, an application attaches the necessary event information to a thread (either using system calls or using a language interface to support events). When a thread makes an invocation, the event information is packaged and sent along with the rest of the per-thread data (such as parameters for invocation, I/O channel information, etc). The information is used by the run-time system/operating system running on the remote machine to deal with events arising for the thread during its visit to the local kernel.

The system provides primitives to raise events to be delivered to threads and objects. The signaler need not be aware of the current location of the thread or the object. In addition, the design also allows for cooperating objects and threads to form an application group: events raised for an application group will be delivered to all members of the group.

When an event is delivered, a handler specified by the recipient will be run. Handlers can be statically bound to an object by specifying them at compile-time. Handlers can also be dynamically bound to an object by attaching them to a thread and invoking the object using the thread. In the latter case, the handlers will be executed in the address space of the invoked object. This makes it possible for the thread to install its own handlers to handle an event, regardless of where it is executing in the distributed system. Such handlers are executed in the address space of the current object in which the thread is active.

Events are raised by the system implicitly (e.g. a page-fault) or explicitly (e.g. a timer notification). User programs can raise events and provide details about the intended recipients. An event can be raised synchronously (signaler blocks) or asynchronously (signaler continues). In the case of synchronously raised events, the fate of the signaler (resume, terminate) is decided by the handler.

The options provided for raising events are outlined in table 1.

<table>
<thead>
<tr>
<th>Call</th>
<th>Recipient of event e</th>
</tr>
</thead>
<tbody>
<tr>
<td>raise(e, tid)</td>
<td>Thread tid</td>
</tr>
<tr>
<td>raise(e, gid)</td>
<td>Threads in group gid</td>
</tr>
<tr>
<td>raise(e, oid)</td>
<td>Object oid</td>
</tr>
<tr>
<td>raise_and_wait(e, tid)</td>
<td>To tid (synchronous)</td>
</tr>
<tr>
<td>raise_and_wait(e, gid)</td>
<td>To gid (synchronous)</td>
</tr>
<tr>
<td>raise_and_wait(e, oid)</td>
<td>To oid (synchronous)</td>
</tr>
</tbody>
</table>

Table 1. Options for raising events.

and termination models), debuggers, monitors and user-level tools (two-phase commit protocols, terminating a distributed computation, etc), and is the basis for the design of our scheme to dynamically replace object references in an application, presented in this paper.

3. The name binding problem

In a typical distributed application, a client establishes a connection with a server using an intermediate name server. Clients query the name server using a symbolic name for the service and the name server returns a name binding for the server. The name binding is a handle that the client uses to directly communicate with the server. A handle is usually an abstract data object that encodes the physical location of the server as well as other information (such as port numbers) necessary to contact the server directly. The handle is usually cached by the client and reused during its (the client's) lifetime.

Problems arise if a cached handle is no longer valid. This could occur due to any of the following reasons:

- The server has been reimplemented (reconfigured), leaving the old interface intact, while adding new capabilities. The reimplemented server may be assigned a new binding (such as a new port number) by the system.
- The server has been moved to a new node, for load balancing or running maintenance operations on the current node.
- New servers have been introduced to handle excessive loading on existing servers.

In the first two cases, the cached handles are no longer valid and object invocations using the handles fail. In the last case, even though the cached handle is still valid, it should be invalidated so that the client may now be bound to a new server. Notice that this type of re-binding is only possible if the server is stateless: i.e., it does not retain any information relevant to the client from a previous service. However, a large class of applications, such as bulletin board services, yellow page servers and on-line encyclopedias, fall under this category.

Clients can always access the 'correct server' by using delayed binding schemes. In order to understand delayed binding, let us briefly review the common binding schemes. A client handle is bound to a server in one of three ways:

- Statically, during program compilation. In static binding the client program completely specifies the server's location and other necessary information (such as port numbers). No name server is necessary in this case.
- At program startup time, by making a call to a name server responsible for managing the binding. The client usually binds to the server prior to first access of the service and the binding information is cached for all subsequent accesses.
- Dynamically, prior to every invocation.

Static binding is too rigid and dynamic binding, in spite of its flexibility, is too costly to use in practice. Hence,
most applications use the second approach: i.e., bind to a server once, and use the cached binding for subsequent access. This is the approach provided by name servers such as the Xerox Clearinghouse mechanism and is essentially similar to the name binding scheme used by the commercial RPC implementations that use portmappers for name-to-port-number bindings [10].

Caching the binding has the usual invalidation problem: if the server information changes due to any of the reasons mentioned above, the bindings are no longer valid. Most implementations consider any change to the server as an error condition and notify the client when an access is made.

Systems that provide replicated servers (such as Circus [9]) provide mechanisms to retry the binding if the cache data is stale. However, these mechanisms do not take into account concurrency within client applications: i.e., the client itself may be a server that is being shared by many, unrelated, applications. In such cases, the binding data may be stored in thread-specific memory (for example, a per-thread stack) and invalidations need to be on a per-thread basis and not per-object basis. Thus notification mechanisms used for invalidations and retrying must take into account the concurrency of applications and act appropriately.

Also, most name servers do not provide support for server replacements or reconfigurations. A new instance of an existing server or a new instance of a server derived from an existing server class, even though type compatible, is flagged as an error by most name servers. This is done to detect and avoid the case of a client incorrectly binding to a different server with the same symbolic name and entry points.

Allowing servers to be migrated to new nodes introduces another problem of maintaining forwarding addresses. For example, in the Emerald system [13] which allows objects to migrate from one node to another both of the nodes maintain information about the new address/location of the object. If a client tries to access the object at the original site, the request is forwarded to the new site and the new address piggy-backed in the reply. This leads to a garbage collection problem, as the old node cannot easily tell when to discard the forwarding address. In addition, the binding information for objects are maintained on a per-node basis, making it impossible to implement shared clients.

The design goals achieved by our proxy update facility are the following:

- Address the problem of object concurrency. This allows applications sharing an object to be using different server instances, at the same time.
- Allow transparent replacements to be made, where desired by the client. Our scheme allows a cached binding to be invalidated and rebound, without the introduction of any special purpose code into the client and server objects.
- Illustrate the generality of the event notification mechanisms that we designed for the shared-object/computation-thread environment.

The design of a general purpose distributed/replicated name service is not addressed by this work. We assume the existence of a directory based hierarchical name service facility such as the Internet Domain Name Service [1] or the name service design specified by Lampson [14] and Cheriton and Mann [6].

4. Dynamic proxy updates

A proxy object is an object (similar to an RPC stub) that is used to provide network transparency and hide argument passing details in a distributed programming environment [22]. For example, assume that a client object \( C_1 \) is using services provided by a server \( S_1 \). The client uses a proxy object, which we call \( S_1\.ref \) (reference to \( S_1 \)), representing the server. \( S_1\.ref \) has an interface which is identical to \( S_1 \) (same typed entry points) and is a part of the client's address space. In addition, a proxy maintains state information (such as current location) about the server. The proxy object masks the complexities of accessing the real server by managing heterogeneity (marshalling/unmarshalling of the parameters of the invocation), providing location transparency (through name servers), etc.

To access the server, the client simply performs a local object invocation on the server's proxy. Since the proxy is a part of the client's address space, the local object invocation is really a procedure call. The proxy performs the actual remote invocation (cross-address space procedure call) and returns the result to the client.

4.1. Problem description

In the above example, if server \( S_1 \) changes (replaced with an equivalent server, perhaps on another node), an object invocation using the existing binding for \( S_1 \) will result in a failure. The dynamic proxy update scheme addresses the problem of invalidating the cached bindings to the server and obtaining new bindings when a server is replaced.

In a nutshell, our solution to the object replacement problem is to detect and invalidate the stale binding and dynamically update the client's proxy on the fly. The asynchronous event notification facility discussed in section 2 is used for the notification and subsequent rebinding of the client to the server. Using mechanisms provided by the event notification facility and a suitable programming environment, the invalidation and updating is done without any intervention by the client or the server program. In addition, using thread-based notifications, proxies are updated on an application basis, rather than on a per-object or per-node basis. Thread-based notification also allows us to manage concurrency within objects quite easily.

4.2. Proxy construction

Before we discuss the proxy update scheme, we outline how proxies handle object invocations. Some of this discussion refers to C++ code and techniques, but it can be easily extended to any object-oriented language framework [2].

As mentioned earlier, a proxy class provides an interface similar to the remote server object that it represents. In our implementation, proxies are automatically constructed by the compiler. The client code
includes a description of the server's interface (a header file) and the proxy is constructed from the server's description. The following code shows the interface for a dictionary server and how the client declares a proxy object for it.

```c
#include <dictionary.h> // Server's description

class dictionary {  
  // Private data
  public:  
    // Public entry points
    void dictionary(); // Constructor
    void lookup(char *word, char *buf);
  
  // Client code
  private:
  
  // Constructor
  dictionary();

  // A segment of the client code.
  
  // dictionary_ref is a proxy for dictionary
  
  // Constructor
  dictionary_ref mydict("webster");

  void lookup(char *, char *); // synchronous
  void lookup(char *, char *); // asynchronous
};
```

The proxy object `dictionary_ref` is constructed automatically from the description for dictionary and contains methods for invoking operations on the real server. The proxy has methods for synchronous as well as asynchronous invocations (where the invoker does not wait for the results). A sample description of a proxy class is shown below:

```c
class dictionary_ref: public baseclass {  
  // Private data
  public:
    // Private data
    void dictionary_ref(); // Constructor
    void lookup(char *, char *); // synchronous
    void lookup(char *, char *); // asynchronous
};
```

Proxy objects inherit from an environment provided `baseclass`. The `baseclass` class contains methods and data common to all proxies, such as, binding data, methods for marshalling and unmarshalling arguments and event handlers for handling common events posted to the client thread. For example, the `baseclass` provides a handler for an event named `INVOCATION_FAILURE`. When the event `INVOCATION_FAILURE` is posted for a thread, the handler specified in the proxy (on which the invocation was attempted) would get called. The default action is for the handler to terminate the thread that caused the failure (for synchronous invocations) or set an error condition on the proxy (for asynchronous invocations). For asynchronous invocations, if the user tries to perform a later `claim`, an error would be returned. The default action can be altered by constructing `customized proxies` using facilities provided by the programming environment.

A proxy object is typically declared within the client object's address space, either within a function block (on the thread's stack), or as the client object's instance data. Objects allowing concurrent access by multiple applications declare proxies on the thread's stack. Thus each thread gets its own copy of the proxy and each proxy could potentially be bound to different instances of the service.

Some specialized proxies are provided by the system through parameters to the proxy constructor. For example, a proxy that automatically rebinds if an invocation fails can be constructed by passing an appropriate code to the proxy's constructor, as shown:

```c
// Proxy that rebinds if invocations fail
dictionary_ref mydict("webster", REBIND);
```

During proxy construction, the constructor code attaches the default handlers for generic events to the thread. For example, for a proxy that performs rebinding on invocation failures, the proxy constructor code executes the following:

```c
// Declare and initialize event data
event (INVOCATION_FAILURE, bind_fail_handler);
// Attach the event to the current thread
attach(e);
```

In the code segment shown above, the proxy constructor declares `e` to be an instance of an event data type and passes the event name `INVOCATION_FAILURE` and a handler name `bind_fail_handler` to its constructor. Next, the event is attached to the current thread. This declares the intent to handle the `INVOCATION_FAILURE` event and specifies that when the event is posted to the thread, the handler named `bind_fail_handler` should be called. Customized proxies can be constructed by inheriting from the proxy class description. For example, default handlers provided by the environment can be overridden by attaching events and handlers to the thread. Since thread handlers are executed in a last-in first-out (LIFO) manner (details are specified in [17]), the user specified handler will be notified before the `baseclass` handlers are notified.

4.3. Proxy binding

A proxy binds to a server it represents, at the time it is instantiated. The proxy's initializer (such as a C++ constructor) is passed the symbolic name (string name) of the server it represents. The proxy initializer code issues a bind request to a name server, with the symbolic name and the class name of the server. The binding returned by the name server is then cached by the proxy and used for accessing the real server.

4.4. Proxy update

Service requests by the client, through the server proxy, may fail when the cached bindings are no longer valid. This could happen due to the following reasons:

(i) The server node is down.

(ii) The remote node has no knowledge of the server (the server has migrated to a new node or has been replaced with a newer server with a different class name).
(iii) The server rejects the call (server capacity exceeded)

When the client issues a service request, the proxy issues a timer request (configurable by the client) and tries the invocation. If the timer fires, the proxy code raises an INVOCATION_FAILURE event for the thread that requested the timer. This takes care of the first case mentioned above. In the next two cases, either the remote node or the remote server raises the event INVOCATION_FAILURE for the thread.

Due to the nature of the heavy-weight ‘computation-threads’ discussed before, the thread that attempts to perform the invocation at the remote site is ‘logically’ the same as the client thread that performed the invocation on the proxy. Thus, raising an event for the thread causes the handler attached to the client thread to be notified.

In the shared-object/computation-thread environment, it is important that the handler code be run using the original thread of control that the client used when performing the invocation. This allows the handler to access the correct ‘thread state.’ For example, if an application is attached to a terminal window at some node, it is important that the handler also be attached to the same terminal window. In general, since different applications may concurrently be active in an object, it is necessary that the handler thread be the same as the client thread that caused the handler to run.

In the case of synchronous invocations, choosing the correct thread to run the handler is an easy task since the invoking thread will be blocked waiting for the completion of the remote (or cross-address space) call. In fact, all that is needed is to return an error condition to the proxy/runtime system. The proxy can then use the client thread to call the name server and try to rebind. No event notification mechanism is needed in this case. However, using a return value will not work in the case of asynchronous invocations where the calling thread is not blocked pending the outcome of the invocation. In this case, an asynchronous notification mechanism that delivers the event to the original thread is required. The asynchronous event notification facility that we designed handles asynchronous invocations and, as a degenerate case, synchronous invocations as well.

Another point to note is that an asynchronous invocation is really a thread fork. The caller forks a thread and the forked thread runs at the invoked entry point while the parent thread continues to run. This implies that INVOCATION_FAILURE events will be posted to the newly forked thread if invocations fail. Even though the newly forked thread has not taken any explicit action to attach any event handlers, attribute inheritance mechanisms defined by the system pass a parent thread’s attributes to the child thread (similar to process attribute inheritance in UNIX). Thus the newly created thread contains information about the events that the parent wishes to handle. In this case, when an INVOCATION_FAILURE is posted for the child thread, the handler implanted by the parent in the address space of the caller object will be notified.

4.4.1. Handler actions

Recall that the handler is a part of the proxy class. Thus the handler code has access to the cached bindings that are a part of the proxy. When the handler gets notified of an invalid binding, it contacts the name server with the invalid binding and tries to rebind. The information supplied in the binding request is shown below:

\[
<\text{class-name, instance-name, type-of-request}>
\]

The above information is used by the name server to return a fresh binding. The type-of-request indicates if this is a first binding or a retry binding (for an alternate server). This information is used by the name server to provide a replacement server for the instance-name specified in the request.

If a fresh binding is obtained by the handler, the original invocation is retried using state information maintained by the proxy. If the binding request fails, an event TERMINATE is raised by the handler, the default action for which is to kill the thread. Of course, the thread may have attached a handler for handling the TERMINATE event, in which case the action is dictated by the handler.

5. A name server for object replacements

Name servers that support symbolic lookups maintain information about a symbolic name (usually the name of an instance) and its locating address. This information is necessary for name lookups and migrations, but not sufficient to support server replacements. A client requesting a replacement service may be granted the binding to an available instance of the same type or to an instance of a subtype of the original type. Preserving the type = subtype equivalence in object-oriented languages, any operation that can be performed on the original type can also be performed on the subtype.

A name server record contains the server's type (class name), a list of instances of the same type and a set of pointers to possible replacement types (sub-classes or derived classes). When an instance of an existing service class is created, it is registered with the name server. The registry information specifies the type name and the instance name. The name server also associates a timestamp with each instance name (the use of the time-stamp is discussed later).

When reconfiguring a server (effectively introducing a new type that is a superclass of the existing type) the administrator provides the new type name as well as the type name of the existing server class. This creates a one-way link from the name server record for the base type to the record for the newly created type. Notice that the base type is not a valid replacement for the newly created type, since sub-type = type transformations are not valid in object-oriented languages. Figure 2 shows the template of a record and figure 3 shows an example.

A request from a client for a binding contains a class name, an instance name, a request type and, in the case of a retry request (issued by a handler), the timestamp associated with the instance the client is currently bound
to. If the client disallows replacement (as indicated by the request type) and the requested instance is not available, the request fails.

If the request type indicates retry and an instance besides the one supplied in the request is available, the binding for that instance is returned. If no other instance is available but the time-stamp for the instance is newer than the time-stamp supplied in the request (indicates that server instance may have migrated), then the new binding is returned. Otherwise, the request fails.

Maintaining up-to-date information requires administrative intervention. In the next few sections, we outline the actions required for migrating a service and introducing new server instances.

5.1. Dealing with server migration and reconfiguration

Migrating an object to a new node requires the updating of the binding data at the name server. Thus the administrator must explicitly perform a name server update operation and change the location information in the binding.

When a server object is reconfigured (retaining the old entry points while adding new capabilities), a new type name is chosen for the object. A new name server record is created and instance(s) of the new type are created and stored in the name server record. Since the new server is backward compatible with the old server, a link is created from the old type to the new type.

The actions taken by the generic proxy code to deal with migration or replacement are listed below.

- Proxy code, during initialization, attaches an event handler for INVOCATION_FAILURE to the current thread.
- Proxy attempts to bind to the real server and, if successful, stores the bindings in its private data.
- When the client performs an invocation on the proxy, the invocation request is stored in its private data.
- Next, the proxy performs any argument packing and tries the remote object invocation, using the current binding.
- If the invocation succeeds at the remote kernel (the bindings are valid), results are returned. If the object could not be located, the remote kernel raises an INVOCATION_FAILURE event to the thread that requested the invocation.
- The handler for the event INVOCATION_FAILURE runs on the invoking object’s address space. The handler issues a bind request to the name server which either returns a new physical binding or an error condition. If an error condition is returned by the name server, the client is notified of the error.
- The proxy updates the physical name binding so that subsequent invocations will use the new object.

5.2. Dealing with replication

Replicated services are useful for load-balancing and increased availability. For read-only services (such as online encyclopedia, yellow pages, obtaining stock quotes, etc), replication is a good and cheap strategy to deal with overloads and failures, since no consistency maintenance is required. We are only addressing binding issues for services that do not maintain state information about the clients. Stateful services require very strong replica management protocols, such as the ones discussed by Cooper [9].

Replicas are introduced by creating a new server instance and adding its binding data to the server’s existing name server record. Load balancing using replicas requires the servers to be preordained to accept only a predetermined number of invocations. If the number of invocations exceeds the capacity, the server raises an INVOCATION_FAILURE event that results in the proxy action discussed above. When the proxy handler attempts the rebind operation, bindings for a new instance, if available, will be returned.

5.3. Dealing with machine failures

If a machine on which a remote service was configured is down, the invocation request may fail even though a replica of the service is available somewhere else on the network. To deal with downed machines, the proxy mechanism uses a timeout scheme and rebinds using the name server if the response does not come back within the timeout period. This has ramifications, however: if the service request is not idempotent, server consistency may be compromised. Thus this scheme may not be suitable for stateful services that are not idempotent (such as a bank account).

5.4. Dealing with multi-threaded clients

As noted before, clients may be concurrently accessed by unrelated applications. Fortunately, concurrency within an object does not complicate the proxy update design. The thread-based event handling primitives combined with the per-thread memory regions supported by the programming environment handle concurrency quite naturally. This permits applications sharing an object to be bound to different instances of the same service at the same time.

The simplest means of handling concurrency is to declare the proxy object in the thread’s stack. Thus each thread gets its own copy of the proxy which at initialization time may bind to different servers.
6. Discussions

The implementation of the object replacement scheme using proxy updates was made easier by the versatility of the asynchronous event notification facility that we built for the Clouds distributed operating system. The main principle behind the design was the recognition that in a passive object-based distributed system, the computational view is one where long-lived threads map the objects (distributed across the network) in and out of their address spaces. Thus a thread's state (and consequently the application's state) is visible in all the objects it touches, making it easier to seamlessly integrate and customize the scattered objects.

Active object-based distributed programming environments (such as the one provided by Mach) also support primitives for task-based and thread-based notifications [4]. For example, on Mach, it is possible to send an exception RPC report to a task port or to a thread port. However, as is typical of active-object based systems, intra-object reporting is done automatically by the system whereas inter-address space reporting is a programmer's responsibility. Hence, a distributed computation running on an active-object based system requires explicit programming by the object programmer to take care of the inter-object notifications.

Our design of events takes care of intra-address space as well as inter-address space by attributing threads with events. The same approach is taken by the designers of the Alpha distributed operating system where they attribute threads with real-time requirements [8]. The information attached to the threads is visible to all objects, at all sites, and can be used by the kernels and the run-time systems running on the individual nodes. On an active-object based system where threads are local (do not cross address spaces) and servers do not execute on behalf of any clients, such information needs to be transmitted as parameters to every object invocation, for every object partaking in the distributed computation.

The richer set of semantics offered by events, compared to simple message send and receive, made it possible for us to implement distributed system services and administration tools rather easily. All parties partaking in an event notification scheme pre-agree on an event and an action sequence. If the action sequence is encapsulated very well, it could take place anywhere in the distributed system. For example, action for an INVOCATION_FAILURE is well encapsulated: retry the binding, retry the invocation request. Thus it is possible for the sequence to be executed from anywhere in the distributed system. Using simple messages to perform chores such as this requires a lot of explicit actions from the invoking object.

6.1. Performance considerations

One of the goals of our design was to minimize the impact of the proxy update mechanisms on normal operations (cached binding data remains valid and no invalidations and rebinds are required). For normal operations, the extra costs paid for the update mechanisms are the following:

(i) The cost of attaching a handler to the thread (system call), at proxy construction time.
(ii) The extra cost due to the thread state that needs to be transmitted when the thread migrates to a new node (during an invocation).
(iii) The cost of installing an event handler at the remote node. Event information passed along with the thread is used by the thread-management system to request event notification, on behalf of the thread. The information maintained on behalf of the thread includes the events of interest to the thread and the locations and addresses of the handlers.

Items (ii) and (iii) listed above are part of the normal invocation mechanism in our environment. Thread state transmitted as part of the object invocation contains parameters for the invocation, I/O handles attached to the thread and any event the thread has requested notification for. While this extra cost makes remote invocations more 'heavy-weight,' the flexibility it provides is the ability to view a distributed application as seamlessly as a single process. Item (i) has negligible costs and it is paid for only once, when proxy gets initialized.

6.2. Observations

Even though the basic paradigm for building distributed applications is different, the commercial RPC implementations can take advantage of the ideas presented in this paper to provide automatic detection and recovery of client handles (object references). Note that in general, automatically retrying an operation is not always good idea—if the service that is being invoked is not idempotent, retrying an operation may lead to an inconsistent state in the server. In our design, we explicitly designated certain proxies to be retry proxies, thus indicating that retry operations may be performed on behalf of this proxy. In the commercial RPC implementations, the same effect can be achieved by either explicitly associating an exception handler to the call or by specifying to the run-time system through the client stubs (such as passing an extra argument to the call or through an environment mechanism such as the call environment supported by CORBA [20]) to automatically retry the code.

6.2.1. Future work

Even though a general purpose multi-threaded event handling facility is not available on commercial platforms such as UNIX RPC environments, we plan to implement the server object replacement facility on UNIX. Currently, RPCs use a simple portmapper mechanism to perform name binding: the server registers itself with the portmapper (running on the same node as the server). The portmapper assigns a dynamic port number to the server. Clients on other nodes contact the portmapper on the server's node and obtain the server's port number and then use the port number for accessing the service directly.

If a server is changed (for example, a crash and a subsequent restart), all previous binding issues by the portmapper are no longer valid. We plan to implement a smart name service scheme that corrects this problem. However, the UNIX signal mechanism is not suitable for implementing proxy updates as discussed in this paper. Our current plans are to implement a simple proxy update
protocol and smarter RPC stubs. RPC stubs, as they are used now, do not support any customization. We plan to implement proxy objects in an RPC environment, which have the RPC stub code embedded in them and are amenable to customization.

6.3. Conclusions
In our experience, distributed programming using shared-objects, computation-threads, and invocations as the building blocks is far superior to the common RPC-based client-server models. Programming a distributed application can be simplified if the distributed system can be given the look-and-feel of a centralized system. This is the basis for the design of the Clouds Distributed Programming Environment.

To address the problem of concurrency and distribution, we designed mechanisms to make thread attributes visible to all objects the threads visit. One such mechanism is the asynchronous event notification facility. Using thread-based event notification as the primary vehicle, our design provides primitives to signal and handle events on an application basis. Using these primitives, we designed a scheme for dynamically updating a client's references to a changing server in a distributed programming environment. The client uses an automatically constructed proxy to access the service and the proxy code manages the task of updating the client's binding to the server.

We also discussed the design of a name server that allows object replacements. Migrating a server to a new node is handled by simply updating the name server data and the clients eventually rebinding to the new location. Load balancing by introducing new instances is handled by creating new server instances and adding their binding data to the existing name server record. If an overloaded server rejects a service request, the client will eventually rebind to a new server. When a server is replaced with an upgraded interface (but remains backward compatible), the name server handles the rebinding requests from older clients, by using sub-type information created by the administrator.

Acknowledgments
We gratefully acknowledge the reviewers for their comments and suggestions on improving this paper. We also thank Gautam Shah, Ranjit John and L Gunaseelan for their helpful comments and suggestions on this work.

References