# Cluster-Based Computing with Active, Persistent Objects on the Web

Frank Sommers
Autospaces, L.L.C.
5118 Village Green
Los Angeles, CA 90016, USA
fsommers@autospaces.com

Shahram Ghandeharizadeh and Shan Gao
Department of Computer Science
University of Southern California
Los Angeles, CA 90089, USA
shahram@cs.usc.edu, sgao@cs.usc.edu

## Abstract

*This paper describes a middleware that enables its target application to dynamically incorporate heterogeneous nodes of a cluster. It distributes the objects of the application across the nodes with the objective to evenly distribute system load. As such, it eliminates the need for a system administrator to control the placement of data. We describe the architecture of the middleware that facilitates object migration and its decision making components. One aspect of this architecture is a negotiation protocol to facilitate migration of objects from one node to another. Finally, we describe an implementation of this middleware using Java and Sun's Jini framework.*

## 1 Introduction

The increasing ubiquitous availability of high bandwidth and powerful processing is allowing network-remote objects to replace, or augment, HTML-based web pages as the Web's central currency[12, 7, 17]. The key benefit of such objects is their ability to carry both data and the semantics of the data, which they expose via their public interfaces. In this environment, remote method calls facilitate object interaction on the network.

In order for an object to service an incoming method call, it must (a) be executing in the address space of a processor, and (b) arrange for the network software layer running on the object's host to direct arriving method call requests to a communication port reserved for the object. In this paper, we term these active objects[2, 3][1]

As a motivating example, consider an airline that exposes each flight as an object on the network. A Flight object encapsulates all data related to the flight, such as the departure and destination cities and times, the current number of available seats, as well as the flight's current status (sitting at an airport or in flight). Remote objects obtain this information via public, remotely available method calls.

When a new Flight object is created, it registers itself with object registries on the network. These registries allow clients to locate Flights, based on some query criteria, such as the departure and destination cities. An object representing a flight reservation system might locate all Flights between two cities, and present this list ordered by departure times.

When a client locates a suitable Flight, it accesses an object registry for a stub representing that Flight on a remote network node. The client uses the stub to invoke methods on the remote Flight object. Thus, stubs act as proxies for the Flights. There is one copy of this proxy for each remote client node referencing a specific Flight object.

The distributed nature of this application enhances performance. To illustrate, assume that 90% of the system workload consists of queries with 10% performing update transactions. If a centralized transaction processing system is connected via a 40 megabit per second (Mbps) connection to the Internet, and each query/transaction exchanges 1,000 bits with this server, its network connection can support at most 42,000 transactions per second.

A clustered environment enables objects representing different flights to dynamically migrate between nodes of a cluster. While all update transactions impacting a flight object must be redirected to the centralized transaction processing system, a node can process queries referencing its objects without contacting the centralized server. Assume that each node is connected via a 1 Mbps connection to the Internet. Each node may perform 945 operations (105 transactions and 840 queries – each transaction now requires 2,000 bits: 1,000 bits from the client to a distributed node plus another 1,000 bits from this node to the transaction processing system at the centralized server). With the centralized server performing 42,000 transactions per second, this

[1] Note that the definition of "active object" in the mobile agent community is often that of an object which moves from one location to another and, once there, obtains a thread of control to execute automatically. The definition used in this paper is one found in the distributed computing literature.

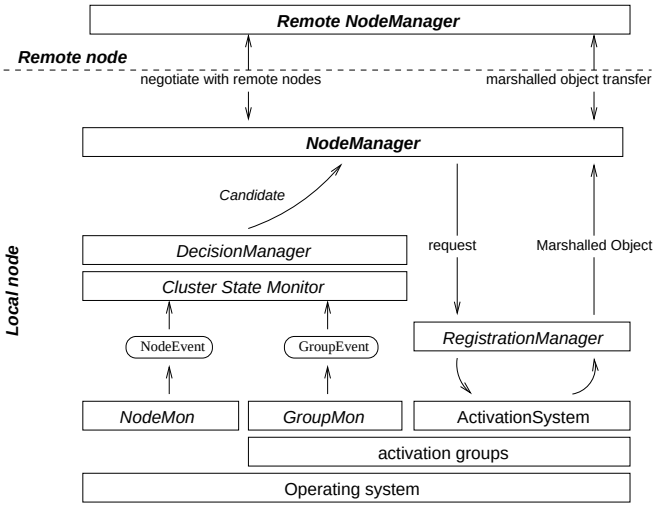


**Figure 1. Middleware architecture overview**

environment can support a workload of 378,000 operations per second before the network bandwidth of the centralized server becomes fully utilized. We would need 400 nodes and an even distribution of the workload across these nodes to attain this objective. Our solution facilitates transparent object migration in support of approximating an even workload across the nodes.

The next section of this paper introduces a framework that facilitates object migration across a cluster of nodes. Section 3 describes a migration protocol nodes might use migrating objects. Section 4 presents an implementation of a middleware using Jini and activatable Java RMI objects. Finally, Section 5 discusses future research directions.

## 2 A middleware solution

The primary motivation for a middleware-based design is ease of use: The only requirement is that the middleware software package be installed on any commodity PC. We assume the presence of heterogeneous hardware and network capabilities in the cluster, where: (a) nodes might differ in their CPU processing speed or available memory, and (b) the network connections between nodes might support a variety of latency and bandwidth characteristics. Thus, it is designed for what we believe is a fairly typical do-it-yourself cluster environment assembled form existing equipment[11]. In addition, we do not assume the presence of a central controller for the cluster: Nodes discover each other's presence and removal in a distributed manner.

In addition, our design strives to accommodate heterogeneity of the active objects. Objects in the system differ in the amount of data they contain, the number of remote method invocations they can service during a given time period, the frequency of incoming method invocations, the amount of data they pass across the network for each method invocation, and especially, the types and numbers of local references an object may hold to other objects. We address this challenge in two ways: First, by allowing the middleware on each node to accommodate pluggable load balancing and migration algorithms, and, second, by having the middleware build a profile for each object, or a graph of objects.

### 2.1 Components of the toolkit

The four logical components of our middleware correspond to the following functionality:

- Monitor the state of the local and remote nodes, and build profiles of objects as well as nodes,
- Decide which objects to migrate and at what point in time based on the node and object profiles,
- Facilitate registration and deregistration of objects on a node,
- Coordinate sharing of node and object profiles, as well as object migration, between nodes.

The overall middleware architecture is illustrated in Figure 1. The main components are the following:
*ActivationSystem*: An interface specifying an object activation system. Our present design utilizes object activation to manage computational resources within a node[20]. Rarely accessed objects become inactive, freeing memory and other finite resources. When a remote method call is directed to an inactive object, an object activator instantiates the object so that it can process the remote method invocation. There is one activation system, per node, which is configured to run as a daemon service.
*RegistrationManager*: Objects interested in object activation must register with a node's activation system. This component facilitates registration and unregistration of objects with the activation system. There is one registration manager per activation system (per node).
*NodeMon*: It collects data about resources on the local node, such as its CPU utilization, available memory or bandwidth. All collected information is configurable. There may be one or more such objects per node.
*NodeEvent*: An event object that transmits pieces of information collected by `NodeMon` objects. Each `NodeMon` produces such events at a frequency that is specified to an implementation of `NodeMon`. `NodeEvents` are multicast on the cluster by `NodeMon` objects.
*GroupMon*: Within each activation system, objects are partitioned into groups. Objects belonging to the same group share runtime characteristics, such as security permissions,

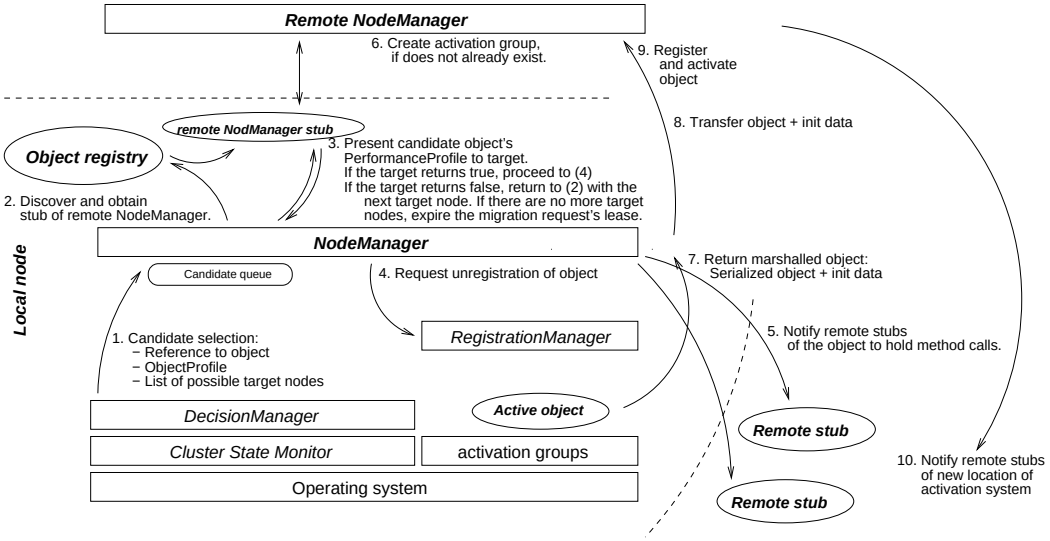


**Figure 2. The object migration protocol**

library load paths, etc. Each group has one `GroupMon` object that collects information about the group, such as the number of objects registered with the group, when objects register or unregister with a group, and when objects become active or inactive in the group. `GroupMon` also collects information about the usage of each object in the group, such as the number of method calls directed to the object, the number of remote references other objects keep to the object via stubs, or the amount of data sent in each method call.

*GroupEvent*: An event object produced by `GroupMon`. When an event of significance occurs in a group, `GroupMon` may fire such an event, encapsulating in it information about the event. `GroupEvent`s are multicast on the cluster by `GroupMon` objects.

*ClusterStateMonitor*: Each node has one instance of this object. It is a consumer of `NodeEvent` and `GroupEvent` objects, and collects their information to build profiles about nodes, groups, and objects not only on the node itself, but on other cluster nodes as well.

*DecisionManager*: It uses information collected by `ClusterStateMonitor` about the local and remote nodes to determine if any objects running on the local node might benefit from migration into another node. This decision is dictated by algorithms which are pluggable into this module. Once a decision is made and a possible node is selected, information about the migration candidate is relayed to `NodeManager`.

*NodeManager*: There is one instance on each cluster node. It negotiates migration of objects with node managers residing on other nodes. If the local node wishes to migrate an object to another node, its `NodeManager` contacts the remote node's node manager, and arranges with it the object's transfer. As well, the local node might be requested to accept an object, in which case `NodeManager` uses `DecisionManager` to decide whether the proposed object should be accepted, given the load characteristics of the local node and the profile of the remote object. `NodeManager` delegates object registration and unregistration with the activation system to the `RegistrationManager`.

*PerformanceProfile*: It encapsulates information about an active object's performance characteristics, such as the number and amount of remote method calls directed to it in a given time period, the average of data passed in method calls, or the amount of CPU resources the object is likely to consume. Each of these pieces of information is contained in a `PerformanceItem`; `PerformanceProfile` is a collection of such items.

Since nodes might have a diverse set of capabilities in a heterogeneous cluster, and objects might have a broad set of profiles, migration decisions are based on rules which are activated on a node as the state of the cluster changes. `DecisionManager` on each node behaves as an expert system: as `Node`- and `GroupEvent`s arrive, `ClusterStateMonitor` continuously asserts these as new "facts" into `DecisionManager`'s memory space (blackboard), resulting in rule activations that match a given system state and object profile.

There are two types of "rules", or decision making algorithms. One migrates objects away from this node, while the other accepts migrating objects from other nodes. Both use the same set of available system state and object profiles. Based on this information, some algorithms might determine that a given object would provide better performance

on another node in the cluster. Such a decision is termed candidate selection.

When a candidate is selected, `DecisionManager` notifies the local object migration coordinator, `NodeManager`, of this fact, informing it of the object to be migrated away from the local node, the remote nodes that are best suited for running the object, based on their state known to `DecisionManager`, as well as the candidate object's performance profile. The latter is relayed in the form of a `PerformanceProfile` object, which includes the pieces of information that prompted `DecisionManager` to select the object as a migration candidate.

`ClusterStateMonitor`, one running on each node, collects `NodeEvents` and `GroupEvents`. `NodeEvents` arrive from both the local node and from remote nodes, whereas `GroupEvents` arrive from both remote and local activation groups. These events are produced by node monitor and group monitor objects, `NodeMon` and `GroupMon`. Each node has one or more node monitor objects, and each activation group has exactly one group monitor object. Information collected by `ClusterStateMonitor` is stored persistently to survive restarts of the node's operating system. Since group and node events are time-stamped, the cluster monitor can store this information in a way that best facilitates comparison between nodes and groups at a given point in time, and this information also facilitates analysis of the cluster's behavior over time.

An implementation of `NodeMon` determines the sorts of data that it is capable of collecting. Each operating system offers different mechanisms for this purpose. On Unix, the /proc virtual file system provides an efficient mechanism to monitor system state. Proc provides access to information from inside the kernel, such as CPU utilization, processes, file system and networking statistics. This is a relatively unintrusive monitoring mechanism, because with most Unix implementations reading the /proc system involves only memory access. On Windows NT, the Windows Management and Instrumentation API (WMI) exposes system information via a COM object[8]. Thus, `NodeMon` objects on Windows would invoke methods on this object. In addition, `NodeMon`s can be implemented to collect information from any third-party tool that facilitates system monitoring[1, 5].

The frequency at which a `NodeMon` object fires `NodeEvent`s is also specific to an implementation of `NodeMon`. In general, event generation should correspond to the frequency of change in the monitored resource's state. Frugality with `NodeEvent`s is important so as not to unduly load the system and the network with event generation and delivery.

We believe that IP multicast is a good way for a node to deliver `NodeEvent`s and `GroupEvent`s to the other nodes in the cluster. An alternative implementation would be to have each node register interest in other nodes' `NodeEvent`s or `GroupEvent`s; `NodeMon` objects, for instance, would then notify event listeners of such events. For our purposes we believe multicast to be more efficient. Another alternative would be to have a central repository, a persistent shared memory, accessible to all nodes, and then all `NodeEvent`s and `GroupEvent`s would be deposited there. Our system, however, avoids reliance on centrally available resources.

### 2.2 An object activation system

Our specification of an object activation system offers four pieces of functionality: (1) Activate objects and receive notifications when objects become inactive; (2) Facilitate easy registration and unregistration of activatable objects; (3) Allow the intercepting of method calls from remote object stubs; and (4) Provide information about activation groups.

These capabilities are further decomposed into interfaces. Providing group information is specified in the `ActivationSystemInfo` interface, while the `ActivationSystemUtil` interface offers a handy registration mechanism, and allows the intercepting of method calls arriving to a registered object during object migration (see Section 3).

`GroupMon` objects register with `ActivationSystemInfo` to listen for `GroupEvents`. `ActivationSystemInfo` produces `GroupEvent`s when a new group registers with the activation system, a new object registers with a group, a group becomes active or inactive, or an object becomes active or inactive. `GroupMon` may decide to aggregate this information at the node before sending out a `GroupEvent` to reduce network traffic (event box carting).

## 3 A consensus protocol for object migration

Since we assume no central controlling entity among the nodes, object migration requires agreement between any two cluster nodes about the transfer of an object. This agreement is facilitated by a protocol consisting of 3 phases. This protocol is outlined in Figure 2.

- Discovery: Locate the new node's `NodeManager` on the network and obtain a reference to it;
- Negotiation: Agree on the transfer of the object with the remote node;
- Transfer: Perform the object's transfer to the new node.

When the local DecisionManager selects a migration candidate, that node's NodeManager receives a migration request (MigrationRequest): It contains a list of potential nodes to migrate the object to, along with a reference to the object, as well as the object's PerformanceProfile. The list of potential target nodes is compiled from the information ClusterStateMonitor collects over time from node and group events arriving from every available node: Based on the object's performance profile, and the state of the other nodes, DecisionManager selects the nodes best suited for the particular object. Due to space-limitations, we do not discuss here decision making heuristics.

The migration protocol requires the mutual consensus of two nodes to transfer and take on an object. Thus, it must account for situations when this sort of agreement is not reached within a reasonable amount of time. When DecisionManager places the migration request in a receiving queue provided by NodeManager, it places a lease on the request. If NodeManager is not able to migrate the object before the lease on the request expires, the migration request will be annulled in NodeManager's incoming queue.

The most common reason for a failure to migrate an object is that other nodes are not willing to accept it. In this case, DecisionManager is free to place another request for the same object's migration in the queue. This operation is idempotent: If an old request is still in the queue, placing another request for the same object's migration has no effect.

The importance of the lease on a migration request is that it accounts for the dynamic nature of the cluster. This is important, because some time will pass between the migration request and when the object is in a state when it can actually be migrated. During this time, the target node's state, such as its load, might change, which might imply that it no longer is the ideal candidate for the object, causing the target node to no longer accept the object when the migration is attempted. If all the potential target nodes reject the object, NodeManager on the object's original host will cause the migration request to expire. At that point, DecisionManager might select a different set of possible target nodes for the object's migration. This mechanism ensures that pending migrations requests that no longer make sense under new conditions of the cluster are not carried out.

### 3.1 Discovery

Once a NodeManager finds a pending request in its queue, it attempts to contact its peer NodeManager on the first potentially suitable node (specified as part of the migration request; see above). This is accomplished through object registries. NodeManager is a remote object: When the activation system starts up on a node, it registers a proxy for its local NodeManager in object registries. Such a proxy includes an attribute value that uniquely identifies its node in the cluster. DecisionManager includes this attribute value in the migration request, facilitating the discovery of a NodeManager proxy.

### 3.2 Negotiation

Once the local node obtains a reference to the remote node's NodeManager, the two nodes can communicate via remote method calls to carry out the negotiation phase of the migration protocol. First, the local NodeManager calls the remote proxy's acceptObject() method, passing the PerformanceProfile of the candidate object as a parameter. The remote node can decide whether it is willing to accept this object, by returning either a true or false value. If it refuses to accept the object, the local NodeManager will perform the lookup and negotiation phases of the protocol from the next possible node indicated by the migration request. If no node accepts the object, the lease on the request will expire (see Section 3).

When one of the target nodes accepts the local object, the NodeManager objects on the two nodes create a communication session. All subsequent method invocations on the remote node will take place in the context of the session. The purpose of the session is to ensure reliable message transmission in the course of moving and registering the object on the new node.

The next step is for the local NodeManager to cause the local object to unregister from the local activation system. A request is directed to the local node's RegistrationManager, which is responsible for registering and unregistering objects with the local activation system, and also for packaging objects ready for shipping to the candidate node.

To unregister, the local object must be in a state when it is not servicing remote method calls. Our design recognizes that an object's implementation is hidden from external view, and that it is up to the object's implementation to determine what are the appropriate times for it to migrate. As well, only the object's implementation would be able to decide what data it needs to operate on new nodes. This data must be available when the object is reconstructed on the new node. To indicate that it is ready for migration, therefore, an object must checkpoint its current state, save instance data to persistent storage, if needed, and must indicate its suitable state for migration. When an object registers with the activation system, the activation system provides a call-back handle, which allows the object both to indicate its readiness for migration, and to provide the data item needed for its reconstruction. When Registra-

tionManager receives this information, the object's migration begins.

### 3.3 Transfer

During object migration, an object's remote stubs must be notified to hold back remote method call forwarding to the object until the object becomes active again on a new node. The activation system on the local node keeps track of remote references to the object based on which remote stubs made method call requests within a certain period of time. The activation system contacts each remote stub with a timeout period for which they need to wait on forwarding remote method calls.

The transfer of the object involves shipping all the data needed for the object to run, and then ensuring that the object is able to initialize (activate) on the new node. Given that an activation group encapsulates information about an object's runtime environment, an activation group with similar characteristics must be present on both nodes. An activation group is created by the activation system based on the group's descriptor. If a group with a descriptor identical to the object's original group descriptor does not already exist in the target node, the original group's descriptor will be shipped to the target. The target node will then attempt to create the activation group. The target node might not be able to satisfy the requirements specified in the group descriptor, causing the migration request to be rejected on the target node.

Once the existence of the proper activation group on the new node is ensured, the object's serialized state and startup data is provided by the local `RegistrationManager` to `NodeManager`, which then passes this information to its peer `NodeManager` on the new node via remote method calls. The remote node then registers the object with its activation system, and causes the object to activate to ensure that it is able to properly initialize. A failure at this point causes the object's migration to be aborted.

When the object becomes active on the new node, the original node passes a list of remote stub references to the new node. The new node's `NodeManager` notifies these stubs with its address. Subsequent method calls made on the stubs will cause the stub to contact the new activation system and obtain from it a live reference to the object.

## 4 An implementation in Jini

Jini[19] is a network inter-operation middleware to help implement highly available, object-based software services. The cornerstones of Jini's design center are lookup services, service proxies, service implementations, and service clients. A Jini lookup service is an object registry that facilitates the locating and retrieving of references to other Jini services. Discovery is the process of finding services, including lookup services, in a Jini federation. As a service discovers lookup services, it registers a proxy object with each. This proxy must be written in the Java programming language, or at least, must be compiled into Java byte code.

Jini service discovery is based primarily on the Java programming language type(s) a proxy implements. In addition, services may optionally specify a set of attributes, or `Entry`s, associated with a service proxy in lookup services. Finally, a service proxy registration results in a globally unique `ServiceID` being issued by the lookup service. A service can store such an ID persistently and reuse it in subsequent lookup service registrations. A Jini federation is illustrated in Figure 3.

Clients can discover proxies that implement a specified set of Java interfaces, a specific `ServiceID`, or any `Entry` attribute. Once a matching proxy is found, the lookup service returns a copy of the proxy to the client. The client thereafter makes local method calls on the proxy.

While Jini proxies do not rely on any particular protocol for network communication, Java Remote Method Invocation (RMI)[15] is a practical, easy-to-use way to communicate between a Jini proxy and remotely available resources on the network. In Java RMI, an object implements an interface that extends the interface java.rmi.Remote, and exports itself at runtime so that it can accept remote method calls. A stub compiler creates a stub object that implements the same Java Remote interface as the object. Once clients obtain a reference to a remote stub, the stub will forward method invocations to the remote object implementation. The RMI runtime facilitates the marshalling and unmarshalling of method parameters and return values.

If a client does not have access to classes required to construct the objects in method parameters, return types, or exceptions, the RMI runtime will automatically download these classes to the client's address space[14]. This is possible because the RMI stub complier annotates a code base location where these classes are available from. This allows the client to construct class loaders directed to load classes from these annotated code-base URLs[6].

An activatable Java object's stub contains two references: A live reference to the remote object, and a reference to the activation system where the remote object is registered. If the first method call to the live reference fails (because the object is unexported), the stub requests the object's activation via the reference to the activation system. If the activation succeeds, the activation system returns a live reference to the object, which the stub will use to proceed with the remote method call. Since the reference to the activation system faults in the live reference, it is called a faulting remote reference[15].

Activatable Java objects are grouped into ActivationGroups when registering with an activation system. An ac-

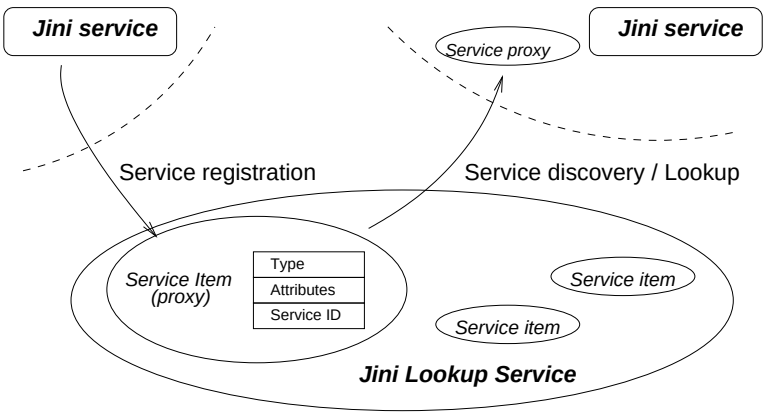


**Figure 3. A Jini federation**

tivation group corresponds to one instance of a Java Virtual Machine (JVM). Thus, object's belonging to the same activation group will be activated in the same JVM. An ActivationGroupDescriptor describes each activation group, and includes such information as the executable for the JVM, the class path and parameters of the JVM, and so on.

When an object's activation is requested, the activation system first checks if the object's activation group already exists. It will then create this activation group, if necessary, starting up the appropriate JVM. It will then construct the object and load it into the group's JVM, passing into it startup data via a special activation constructor. Finally, the activation system returns a live reference for the object to the stub. The Java activation protocol is illustrated in Figure 4[13].

Our system provides an implementation of an activation system specified by Java RMI, which we enabled with the load monitoring and registration capabilities outlined in Section 2.2. Our middleware requires that each cluster node run this activation system as a daemon process. In addition to the class library implementing the middleware in Java, this is the only addition to a standard JDK environment.

In our implementation, `NodeMon` is an activatable RMI remote object, and is also a Jini service that registers with Jini lookup services on the network. The objects that register with our activation system must follow the semantics described in the RMI Specification for active objects, including the requirement that they have a special activation constructor[15].

## 5 Future research directions

Our system's main goal was to construct a facility for the study of dynamic, runtime distribution of objects in a cluster. Our next important step will be to identify algorithms suitable for the runtime placement of objects with diverse characteristics, building on earlier work on data placement in object databases[4].

The existing framework can be extended to support the redundant deployment of Jini services as well. In such a scenario, the original node would simply deactive the object prior to transfer to a new node, and the remote stub would now have 3 references: One to the live object (primary copy), one to the primary node's activation system, and one to the secondary node's activation system. It is important that there be only one primary copy of an object throughout the cluster. Therefore, the middleware must ensure that all stubs contain the same information about the primary.

An object might require resources, such as file handles, I/O ports, or native code, which may not be available on all cluster nodes. The Rio[9] system, developed by Sun Microsystems, allows a Jini service to describe such requirements, and provides facilities for a node to describe its available resources.

The absence of a central controlling entity raises important issues in terms of the cluster's management. We believe that the dynamic discovery of cluster nodes should also extend to administrative tools. The Jini Place API[18] facilitates dynamic incorporation of a node into management modules: For instance, adding a node to the cluster would cause a management module to automatically incorporate this new node into a management console.

We also propose two extensions to the Java Virtual Machine: First, the VM should have an API-level access layer to its utilization and load information, similar to the Windows Management and Instrumentation API. The second extension is to propose an object defined in the Java programming language that corresponds to an instance of the JVM. Such an object could be termed a `Process`, and would provide programming access to managing processes - a functionality similar to process management in the Unix

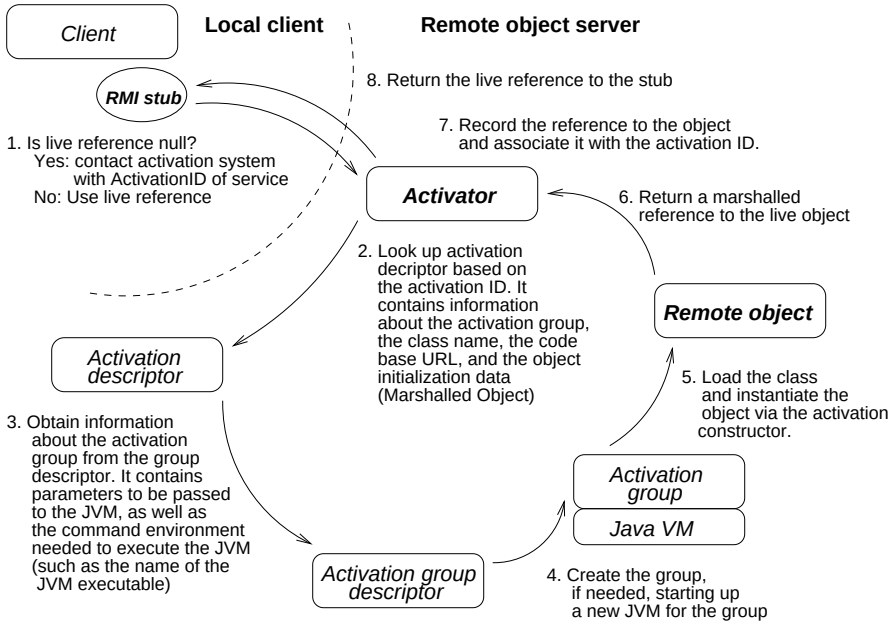


**Figure 4. The RMI object activation protocol**

or Windows NT operating systems. The Real Time Java Specification[16] addresses some of these needs, and a future implementation of this middleware, based on that version of the Java Virtual machine, might take advantage of these capabilities. As well, we believe that the non-blocking I/O subsystem[10], introduced in the 1.4 version of the Java Development Kit, promises to offer significant benefits in reducing the time it takes for an object to migrate across nodes.

For more information on this work, please refer to our project's Web page: *http://perspolis.usc.edu/users/ribbet/*

## References


[1] J. Astalos and L. Hluchy. Cis – a monitoring system for pc clusters. In *Proceedings of EuroPVM/MPI2000, Lecture Notes in Computer Science 1908*, 2000.

[2] M. Brown and M. Najork. Distributed active objects. *Computer Networks and ISDN Systems*, 28(7-11):1037–1052, 1996.

[3] M. Brown and M. Najork. Distributed active objects. Technical Report SRC 141a, Digital Equipment Corporation, Inc., Palo Alto, CA, 1996.

[4] S. Ghandeharizadeh, D. Wilhite, K. Lin, and X. Zhao. Object placement in parallel object-oriented database systems. In *Proceedings of the 10th International Conference on Data Engineering*, 1994.

[5] Z. Liang, Y. Sun, and C. Wang. Clusterprobe: An open, flexible and scalable cluster monitoring tool. In *Proceedings of the International Workshop on Cluster Computing 1999*, 1999.

[6] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Sun Microsystems, Inc, 1998.

[7] V. Machiraju, M. Dekhil, M. Griss, and K. Wurster. E-services management requirements. In *Technologies for E-services Workshop*, Cairo, Egypt, Sept 2000.

[8] Microsoft Corporation. *Windows Management and Instrumentation API*.

[9] S. Microsystems. Rio architecture overview, v1.0. Technical report, Sun Microsystems, Inc., 2000.

[10] S. Microsystems. *New I/O APIs*. Sun Microsystems, Inc., 2001.

[11] G. F. Pfister. *In search of clusters*. Prentice Hall, second edition, 1998.

[12] D. Plummer and D. Smith. E-services: Are they really the next 'e'? Technical Report Vol. 2000, Gartner Group, 2000.

[13] F. Sommers. Activatable jini services. *JavaWorld*, September 2000.

[14] F. Sommers. Object mobility in the jini environment. *JavaWorld*, January 2001.

[15] Sun Microsystems, Inc. *The Java Remote Method Invocation Specification*, 1998.

[16] Sun Microsystems, Inc. *The Real Time Specification for Java*, 2000.

[17] B. Venners. Object versus document: Comparing two ways that software can interact with software. *JavaWorld*, June 2000.

[18] B. Venners. *Jini Place API Draft Specification, Version 0.6*. Artima Software/The Jini Community, 2001.

[19] J. Waldo et al. *The Jini Specification.* Addison Wesley, 1999.

[20] A. Wollrath, G. Wyant, and J. Waldo. Simple activation for distributed objects. Technical Report SMLI TR-95-46, Sun Microsystems, Inc, Nov 1995.