Using Java Interaction Proxies for Web-based Steering of Distributed Simulations

(Draft Paper)

Rajeev Muralidhar and Manish Parashar
Department of Electrical and Computer Engineering and CAIP Center, Rutgers University,
94 Brett Road, Piscataway, NJ 08854.
Tel: (732) 445-5388 Fax: (732) 445-0593 Email: {rajeevdm, parashar}@caip.rutgers.edu

ABSTRACT

This paper presents the design, implementation and evaluation of Java based interaction proxies to distributed computational object to enable web-based steering of distributed simulations. The interaction proxies are part of a distributed object infrastructure for computational interaction and steering aimed at transforming traditional batch simulations into interactive ones by closing the loop between the user and the application. This infrastructure addresses three key issues: (1) Definition and deployment of *Interaction Objects* that encapsulate sensor and actuators for interrogation and control. These objects can be distributed (spanning many processors) and dynamic (be created, deleted, changed or migrated), and can be derived from existing computational data-structures. (2) Definition of a control network interconnecting the interaction objects to enable their discovery, interaction and control and manage dynamic object creation, deletion, and migration. (3) Definition of an *Interaction Gateway* that enables remote access to the application using Java interaction proxies (mirrors). This paper focuses on the definition of these proxies and their remote access. The proxies are created using interaction interfaces exported by the computational objects and use JNI (Java Native Interface) to directly access these interfaces. Access to the proxies uses Java's Remote Method Invocation techniques. The presented research is part of an ongoing effort to develop and deploy a web-based computational collaboratory that enables geographically distributed scientists and engineers to collaboratively monitor, and control distributed applications. Its goal is to bring large distributed simulations to the scientist/engineers desktop by providing collaborative web-based portals for interaction and control.

I. INTRODUCTION

This paper presents the design, implementation and evaluation of Java based interaction proxies to distributed computational object to enable web-based steering of distributed simulations. The presented research is part of an ongoing effort to develop and deploy a web-based computational collaboratory that enables geographically distributed scientists and engineers to collaboratively monitor, and control distributed applications. Its goal is to bring large distributed simulations to the scientists'/engineers' desktop by providing collaborative web-based portals for interaction and control.

Simulations are playing an increasingly critical role in all areas of science and engineering. As the complexity and computational costs of these simulations grows, it has become increasingly important for the scientists/engineers to be able to monitor the progress of these simulations, and to control or steer them at runtime. The utility and cost-effectiveness of these simulations can be greatly increased by transforming the traditional batch simulations into more interactive ones. Closing the loop between the user and the simulations enables the experts to drive the discovery process by observing intermediate results, by changing parameters to lead the simulation to more interesting domains, play what-if games, detect and correct unstable situations, and terminate uninteresting runs early. For example, the scientist can modify a boundary condition based on the current state of the simulation. Similarly in a simulation based on adaptive meshes, additional resolution can be added on the fly in regions that need it. Using

checkpoint/restart capabilities, the experts can now stop, rewind, and replay simulation time step(s) with different parameters when it does not correspond to expected values. We believe that interrogation and interaction capabilities will transform simulations into true research modalities.

Enabling seamless interaction and steering high-performance parallel/distributed applications presents many challenges. A key issues is the definition and deployment of interaction objects with *sensors* and *actuators* {[9], [10]} that will be used to monitor and control the applications. These sensors and actuators must be co-located with the computational data-structures and must encapsulate the modes of interaction of the object. Defining these interfaces in a generic manner and deploying them in distributed environments can be non-trivial, as computational objects can span multiple processors and address spaces. The problem is further compounded in the case of adaptive applications (e.g. simulations on adaptive meshes) where computational objects can be created, deleted, modified and redistributed on the fly. Another issue is the deployment of a control network that interconnects these sensor and actuators so that commands and requests can be routed to the appropriate set of computational objects, and information returned can be collated and coherently presented. Finally, the interaction and steering interfaces presented by the application needs to be exported so that they can be easily accessed by a group of collaborating users to monitor, analyze, and control the application.

This paper presents the design, implementation and evaluation of Java based interaction proxies to distributed computational object to enable web-based steering of distributed simulations. The proxies are part of an interactive object infrastructure that addresses three key issues:

1. Definition and deployment of interaction objects that encapsulate sensor and actuators for interrogation and control. Interaction objects can be distributed (spanning many processors) and dynamic (be created, deleted, changed or migrated), and can be derived from existing computational data-structures. Traditional (C and Fortran) data-structures can be transformed into interaction objects using C++ wrappers as described in the paper.
2. Definition of a scalable control network interconnecting the interaction objects to enable discovery, interaction and control of distributed and dynamic computational objects and manage dynamic object creation, deletion, and migration.
3. Definition of an *Interaction Gateway* that enables remote access to the application using Java interaction proxies (mirrors). The proxies are created using interaction interfaces exported by the interaction objects and use JNI (Java Native Interface) [1] to directly access these interfaces. Access to the proxies uses Java's Remote Method Invocation (RMI) [17] techniques.

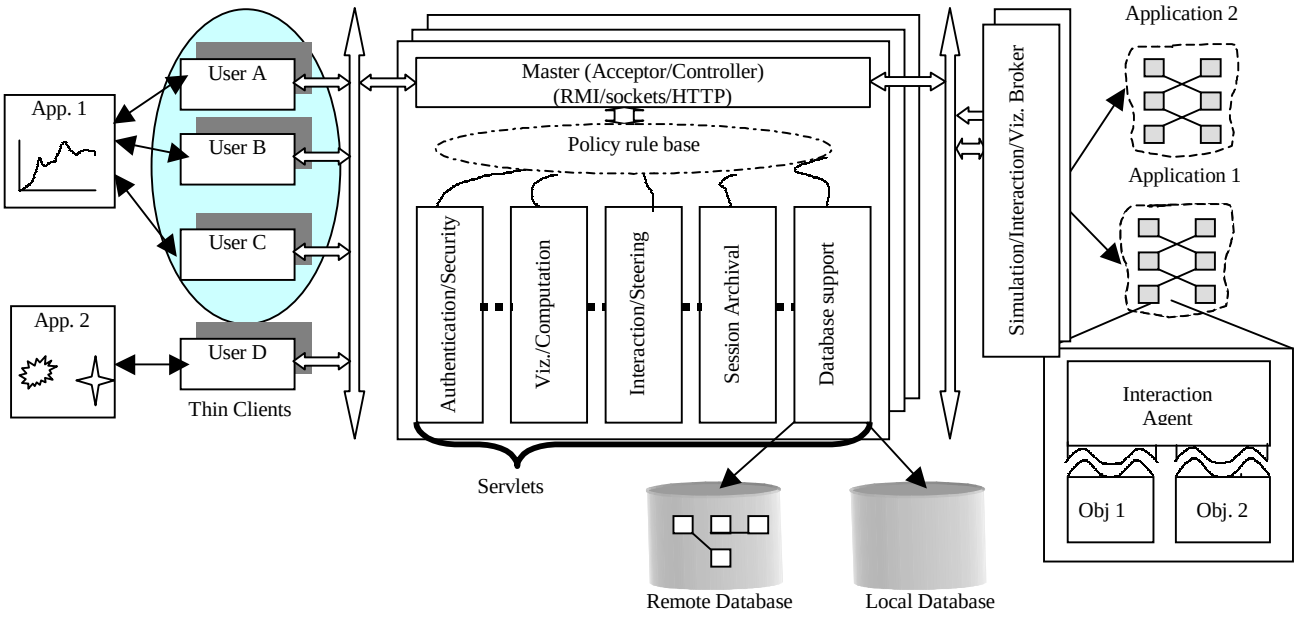
The research presented in this paper is part of DISCOVER (*D*istributed *I*nteractive *S*teering and *Co*llaborative *V*isualization *E*nvi*r*onment), an ongoing research initiative aimed at developing a web-based interactive computational collaboratory. The current implementation of DISCOVER enables geographically distributed clients to use a web-based portal to simultaneously connect to, monitor and steer multiple applications in a collaborative fashion. More information about the DISCOVER project can be found at *http://www.caip.rutgers.edu/TASSL*.

The final paper will be organized as follows. Section II presents a brief introduction of the DISCOVER architecture. It describes the formulation of interaction objects, interaction agents and the control network. Section III describes the JNI-based Interaction Gateway. Section IV presents an overview of the implementation. Section V presents an experimental evaluation of the infrastructure. Section VI describes the related work. Section VII presents concluding remarks and outlines future work. Due to space constraints, we present only Sections II, III, and VII in this abstract.

II. DISCOVER: AN INTERACTIVE COMPUTATIONAL COLLABORATORY

Figure **1** presents an architectural overview of the DISCOVER collaboratory aimed at enabling web-based interaction and steering of high-performance parallel/distributed applications. The system conforms to a 3-tier architecture composed of detachable thin-clients at the front-end, a network of interaction Web servers in the middle providing seamless access to control and collaborate over multiple applications, each composed of a control network of sensors/actuators and interaction agents at the back-end. The

front-end consists of a range of detachable clients, from palmtops connected through a wireless link to high-end desktops with high-speed links. Clients can connect to a server at any time using a browser to receive information about active applications. Furthermore, they can form or join collaboration groups and can (collaboratively) interact with one or more applications based on their capabilities. Information displayed at the client (e.g. surface/line plots, data, text) is updated on demand, or automatically as the application evolves. The client interface supports two desktops – a local desktop and a virtual desktop. The local desktop represents the users private view, while the virtual desktop is a metaphor for the global virtual space through which multiple users can collaborate with the aid of tools like whiteboards and chats. Application views (e.g. a plot) can be made collaborative by creating them in the virtual desktop or transferring them from the local to the virtual desktop. Session management and concurrency control is



based on capabilities granted by the server.

Figure 1 - Architectural Schematic of the DISCOVER Interactive Computational Collaboratory

A. Interaction Objects for Interrogation and Control

Interaction objects are objects that encapsulate sensors and actuators and can support interrogation and control. An interaction object exports a set of *View* and *Command* interfaces which define the modes of interaction for the object. These interfaces are published and can be externally accessed. View interfaces represent sensors and define the type for information that the object can provide in response to an interrogation. For example, a *Grid* object might export views for its structure and distribution. Similarly, a *GridFunction* (application field defined on a grid) object might export views such as iso-surface plots, norms, maximum/minimum values, etc. Views might be on-demand or persistent, i.e. they are continually monitored and automatically updated. Commands represent actuators and define the type of controls that can be applied to the object. Commands for the *Grid* object may include refine, coarsen, and redistribute. Similarly, those for the *GridFunction* may set or reset its data values. Views are associated with viewers at the interaction/steering front-end that can process the views and present them in the appropriate manner.

Interaction objects can be classified based on the address space(s) they can span during the course of computation as *local*, *global*, and *distributed objects*. Local interaction objects are locally created by a computational node. These objects may migrate to another processor during the lifetime of the application, but exist in a single processor's address space at any point of time. Multiple instances of a

local object could exist on different processors at the same time. Global interaction objects are similar to local objects, except that there can be exactly one instance of the object (across all processors) at any time. A distributed interaction object spans multiple processors' address spaces. An example is a distributed array partitioned across available computational nodes. These objects contain an additional *distribution* attribute that maintains the its current distributed type (blocked, inverse space filling curve-based, or custom) and layout. This attribute can change during the lifetime of the object as the object is redistributed. Like local and global interaction objects, distributed objects can be dynamically created, deleted, or migrated.

B. A Control Network for Interaction and Steering

The control network has a hierarchical "cellular" structure with three components as shown in Figure 2. Computational nodes are partitioned into *interaction cells*, each cell consisting of a *Discover Agents* and a *Base Station*. The number of node per interaction cell is programmable. Discover Agents are present on each computational node and manage the interaction objects on the node. The Base Station manages interaction objects for the entire interaction cell. It may or may not be a dedicated processor (it could be a high-priority process executing along with the computation). The highest level of the hierarchy is the *Interaction Gateway* that provides a Java-enabled interface to the entire application. The cellular control network is automatically configured at run-time using an underlying messaging environment and information about the available number of processors.

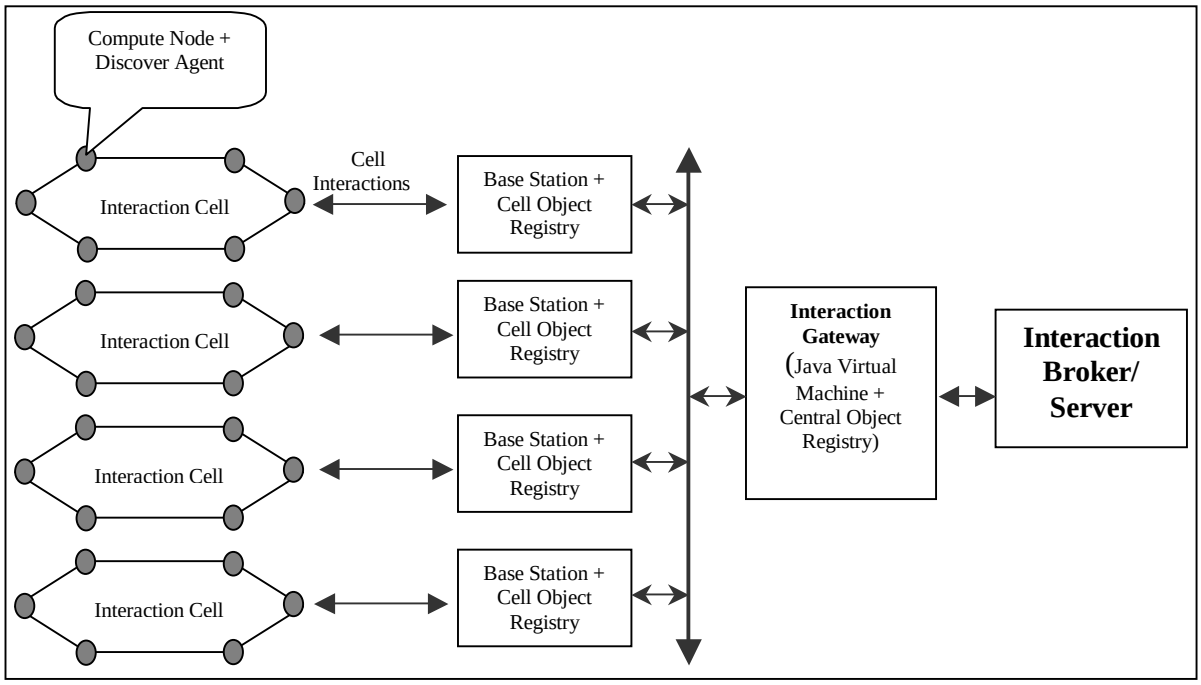


Figure 2: Control Network of Computation and Interaction Nodes within the application

1. Discover Agents and Base Stations

Every computation node houses a *Discover Agent* (DA) that maintains a *local object registry* of references to all interaction objects currently active and registered by that node and manages interactions with these objects. The Discover Agents export the details of the object registration, which include the name with which the object was registered and its type (local, global or distributed) to the respective Base Stations using an interaction IDL (Interface Definition Language). Interaction IDL is a simple custom-built language to describe interaction information of an interaction object. *Base Stations* form the next level of control network hierarchy. They maintain interaction object registries for an entire interaction cell and export these to the Interaction Gateway. At startup, each Base Station configures its interaction cell

using information about the cell size and available number of processors. It then creates a *cell object registry* containing a list of interaction objects and their interaction information exported by each of the DA's in its cell, and exports this to the Interaction Gateway. During interaction, all responses and updates from the DA's are collected by the Base Station and forwarded to the Interaction Gateway.

Interaction object migrations are monitored by the DAs and Base Stations to ensure that the object references are always valid. If the migration is within the interaction cell, the DAs involved will handle the migration and then notify the Base Station of the change. If migration is across cells, the respective Base Stations manage the migration and the Interaction Gateway is updated.

2. Interaction Gateway

The Interaction Gateway represents an interaction proxy for the entire application. It stores information about all the interaction objects currently active and registered by the application and, is responsible for interfacing with external interaction servers, delegating interaction requests to the appropriate base stations and discover agents, and for combining and collating responses. This is explained in detail in the following section.

III. JAVA-BASED INTERACTION GATEWAY

The Interaction Gateway hosts a *central object registry* containing the interfaces of all interaction objects exported by the Base Stations, and creates Java mirrors for each of these objects using the Java Native Interface (JNI) [1][18]. The Java mirrors are registered with a RMI (Remote Method Invocation) [17] registry service also executing at the Interaction Gateway. This enables an external Interaction Server to gain access to and control the interaction objects using the Java RMI API. The definition of the interaction proxies and their remote access is described below.

A. Java Interaction Proxies

The Java proxy (or mirror) of an interaction object contains only its exported interaction information and functions as a proxy to the actual interaction object residing on a remote address space. It uses the *proxy pattern* [13] to create a placeholder for the remote (possible distributed) interaction object (the subject) and control access to it. It uses the JNI API to accomplish this. The Java mirror object definitions are created for all interaction objects during application development. The process may be performed manually or may be automated using lightweight tools. We are currently looking at the TwinPeaks [20] technology that uses a similar approach to automate the conversion of C++ classes to Java classes.

Analogous to the virtual base classes provided by the object management library for creating interaction objects, all mirror objects are derived from a corresponding Java base class. A Java `Interface` specifies the methods exported by the base class, which extends from the `UnicastRemoteObject` class of the RMI package, thus enabling the registration of any derived class with the RMI registry executing on the Gateway. Creation of a Java mirror for an interaction object consists of the following steps:

1. For every interaction object that needs to be created and registered by the application, a Java `Interface` is defined by extending from the `java.rmi.Remote` interface. This interface consists of the interaction methods (Views/Commands) exported by the corresponding object with the same names and analogous signatures. This step is in accordance with the standard procedure for creating remote server objects using RMI. The remote interface is simply a list of methods that the Interaction Server can invoke on the (mirrored) interaction objects.
2. The Java mirror is created by implementing a class that extends from the base class mentioned earlier. The mirror implements the View/Command interfaces being exported by the interaction object using JNI to invoke native code. Essentially this involves a sequence of code that accomplishes the following:
    a. Compose an interaction request for this method using the interaction IDL.

b. Using JNI, invoke a native method of the Interaction Gateway to process this interaction request.
c. Wait until a response arrives from the Gateway. As mentioned earlier, interaction request processing will involve: (1) a lookup in the object registry of the Gateway, (2) message(s) to the relevant Base Station(s) and Discover Agent(s) to route the interaction request to the correct compute node(s), (3) processing the request at the compute node, (4) receiving the response at the Gateway (and possibly collating messages in the case of distributed objects) and (5) a final JNI call from the Gateway to copy the response into a member variable of this class.
d. Return the response. RMI will ensure that the response is returned to the Interaction Server by suitably marshalling it.

B. Java mirrors and the RMI Registry

The Java mirror objects are instantiated by the Interaction Gateway when the Central Object Registry is created. The Gateway uses JNI to instantiate corresponding Java mirrors for every interaction object registered. The mirrors are bound to the RMI Registry with the unique names provided by the application at the time of object registration. These names have to be known at the Interaction Server in order that the latter can subsequently make use of RMI to invoke necessary interaction request methods. This is accomplished in a simple manner. A Java interface is defined containing a single method to acquire the list of names by which interaction objects (more specifically, their Java mirrors) are bound with the RMI registry. The object implementing this interface is bound with the RMI Registry using a well-known name. When the application is registered with the server, the server can acquire the list of interaction objects registered with the RMI registry using this well-known service. The server can then use the Reflection API [19] to publish interaction information (details of the View/Command interfaces exported) of the registered interaction objects to the collaborating clients. During interaction, the server can use RMI to directly invoke corresponding methods on the (mirrored) interaction objects. Interaction requests are processed by the mirrored interaction objects as described before.

C. Interaction Sequences

Figures [5] and [6] depict typical interaction scenarios that can occur and the flow of information in the control network hierarchy. In Figure [5], `oilwell` is a local, non-distributed interaction object created on compute node 1 that belongs to Interaction Cell 1and is registered with Base Station 1. Its Java mirror is created at the Gateway and registered with the RMI registry with a unique name. When the server invokes a method (View in this case) on the mirror object using RMI, the mirror invokes a method on the Central Object Registry to perform the necessary sequence of steps involved in request processing. The Central Object Registry performs a look up for the `oilwell` object. Its entry in the registry indicates that it was exported by Base Station 1. Hence, the view request is sent to Base Station 1, which performs a similar lookup in the Cell Object Registry. This indicates that the `oilwell` object was exported by Compute Node 1. The request is again forwarded to Compute Node 1, where the Interaction Agent receives the request and invokes the `oilwell` object's `processMessage()` is invoked. The generated response is packaged using the same Interaction Description Language and exported to the Gateway. Here, the Gateway uses JNI again to pass on the obtained response to the mirror object, from where the response is exported to the server, using the standard method invocation procedure involved in RMI.

In Figure [6], there are two Interaction Cells each consisting of eight nodes. The situation is similar except that the object now in consideration is distributed across 2 compute nodes – node 1 and node 8. The corresponding Java mirror is similarly created by the Gateway and registered with a unique name with the RMI registry. The sequence of steps followed to invoke an interaction request on the `GridHierarchy` object (and its mirror) is as mentioned earlier. In this case, however, when a lookup is performed at the Gateway for the object, it is found to be a distributed object and had been registered by 2 base stations. Hence this interaction request is broadcast to both the base stations. Subsequently, compute

nodes 1 and 2 independently generate the `XYSlice` requested on the `GridHierarchy` object. Both the responses are sent to the base stations and further up the hierarchy to the Gateway, where a pre-registered callback function `GridHierarchy::gather()`is invoked on the independently generated data. This function assembles the individual pieces of data by performing a reduce operation on the data sets. The response is now ready to be sent out to the Web Server.

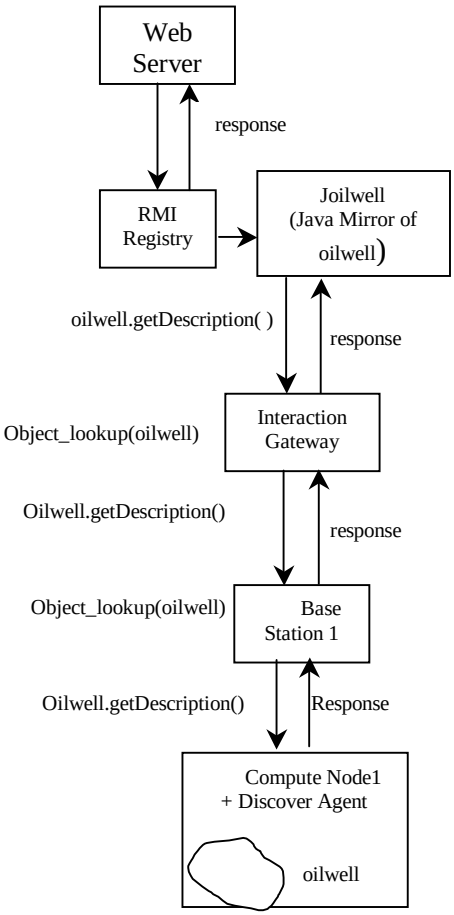


Figure 5: Processing a View Request for a Non Distributed Object

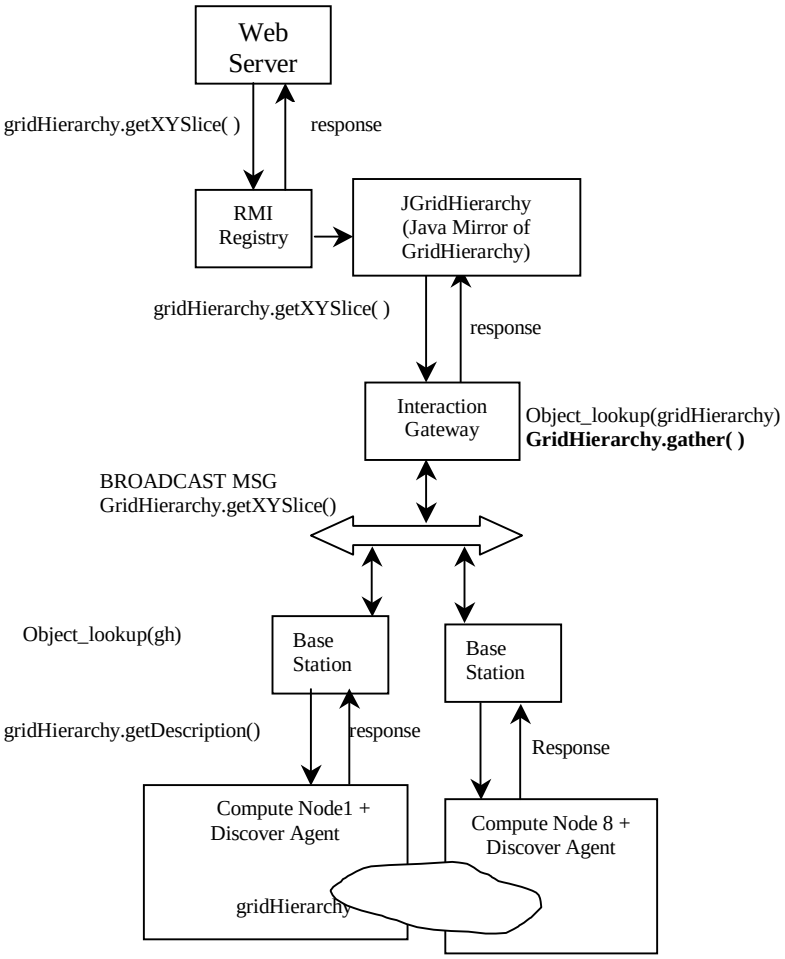


Figure 6: Processing a View Request for a Distributed Object

IV. RELATED WORK

A number of systems have been built for interactive steering and control of high performance applications. Only systems that are similar in architecture and implementation are included in this section. In the *Gateway* [3] system, multiple clients can connect through a CORBA-based [16] middle-ware to a back end consisting of high-end computing resources managed using the *Globus* [4] meta-computing toolkit. In the *Mirror Object Steering System* [5][15], a Mirror Object Model is used to translate data structures into *mirror objects*, which are CORBA-style objects that can then be used for monitoring and steering. In the *CSE* [6], multiple existing applications can be integrated into the framework by annotation of the application source code. The applications and multiple user interface processes can be distributed over different platforms. In *Falcon* [7], existing applications can also be integrated by manual annotation of application source code and provides the end user with an abstraction that reveals only the important steering parameters and output data, which are defined and exported by the application. *Progress* [8] extends the steering methods of Falcon implements steering through *actuators* which operate on *steering*

objects and instrumentation points are explicitly inserted in the application program at compile time. *SCIRun* [11] uses visual programming to create a steering application and is suited for developing new applications for interactive steering. It allows for some monitoring of a running application and limited modification of the data flow network that constitutes the running application. It also provided a visualization interface for observing application data continuously. *Autopilot* [12] provides additional support for automated performance steering in systems where decisions are generated by fuzzy logic and neural networks to help the system scale.

V. REFERENCES

[1]. Java Native Interface Specification, *http://web2.java.sun.com/products/jdk/1.1/docs/guide/jni*.

[2]. John A. Wheeler et al., "*IPARS: Integrated Parallel Reservoir Simulator*", Center for Subsurface Modeling, University of Texas at Austin, *http://www.ticam.utexas.edu/CSM*.

[3]. Bill Asbury, Geoffrey Fox, Tom Haupt, Ken Flurchick "*The Gateway Project: An Interoperable Problem Solving Environments Framework for High Performance Computing*", *http://www.osc.edu/~kenf/theGateway*.

[4]. I. Foster, C. Kesselman, "*Globus: A Metacomputing Infrastructure Toolkit*," International Journal of Supercomputer Applications, 11(2): 115-128, 1997.

[5]. Greg Eisenhauer, K. Schwan, "*Mirror Object Steering System*", *http://www.cc.gatech.edu/systems/projects/MOSS*.

[6]. Robert van Liere, Jan Harkes, Wim de Leeuw. *A Distributed Blackboard Architecture for Interactive Data Visualization*. Proceedings of IEEE Visualization'98 Conference, D. Ebert, H. Rushmeier and H. Hagen (eds.), IEEE Computer Society Press, 1998.

[7]. "*The Falcon Monitoring and Steering System*", *http://www.cs.gatech.edu/projects/FALCON*.

[8]. J. Vetter and K. Schwan, "*Progress: A Toolkit for Interactive Program Steering"*, Proceedings of the 1995 International Conference on Parallel Processing, pp. 139-149. 1995.

[9]. W. Gu, J. Vetter, K. Schwan, "*Computational Steering Annotated Bibliography*", SIGPLAN Notices, 32 (6): 40-4 (June 1997).

[10]. B. Schroeder, G. Eisenhauer, K. Schwan, J. Heiner, P. Highnam, V. Martin and J. Vetter (1997), "*From Interactive Applications to Distributed Laboratories*".

[11]. S.G. Parker, C.R. Johnson. *SCIRun: A scientific Programming Environment for computational steering*. In Proceedings of Supercomputing '95, 1995.

[12]. Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, and Daniel A. Reed, "*Autopilot: Adaptive Control of Distributed Applications*," *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[13]. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "*Design Patterns*", Addison Wesley Professional Computing Series, 1994.

[14]. MPI Forum. MPI: Message Passing Interface. Technical Report CS/E 94-013, Department of Computer Science, Oregon Graduate Institute, March 1994.

[15]. Greg Eisenhauer, "*An Object Infrastructure for High-Performance Interactive Applications*", PhD thesis, Department of Computer Science, Georgia Institute of Technology, May 1998

[16]. "*CORBA: Common Object Request Broker Architecture*", *http://www.omg.org*.

[17]. Remote Method Invocation, *http://www.javasoft.com/products/jdk/rmi/index.html*.

[18]. Java Native Interface Invocation API, *http://www. javasoft.com/books/tutorial/native1.1/invoking/index.html*.

[19]. Java Reflection API, *http://www.sun.com/products/jdk1.2/docs/guide/reflection*.

[20]. TwinPeaks, *http://www.javaworld.com/javaworld/jw-05-1996/jw-05-twinpeaks.html*