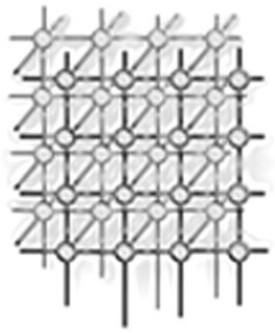
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **15**:957–977 (DOI: 10.1002/cpe.741)

A distributed object infrastructure for interaction and steering



Rajeev Muralidhar and Manish Parashar*,†

The Applied Software Systems Laboratory, Department of Electrical and Computer Engineering, Rutgers, The State University of New Jersey, 94 Brett Road, Piscataway, NJ 08854, U.S.A.

SUMMARY

This paper presents the design, implementation and experimental evaluation of DIOS (Distributed Interactive Object Substrate), an interactive object infrastructure to enable the runtime monitoring, interaction and computational steering of parallel and distributed applications. DIOS enables application objects (data structures, algorithms) to be enhanced with sensors and actuators so that they can be interrogated and controlled. Application objects may be distributed (spanning many processors) and dynamic (be created, deleted, changed or migrated). Furthermore, DIOS provides a control network that interconnects the interactive objects in a parallel/distributed application and enables external discovery, interrogation, monitoring and manipulation of these objects at runtime. DIOS is currently being used to enable interactive visualization, monitoring and steering of a wide range of scientific applications, including oil reservoir, compressible turbulence and numerical relativity simulations. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: distributed and dynamic objects; computational interaction and steering; sensors and actuators; computational collaboratory; scientific applications; parallel and distributed computing

1. INTRODUCTION

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 important for scientists and engineers to be able to monitor the progress of the simulations and to control and steer them at runtime. The utility and cost-effectiveness of simulations can be greatly increased by transforming traditional batch simulations into more interactive ones. Closing the loop between the user and the

*Correspondence to: Manish Parashar, The Applied Software Systems Laboratory, Department of Electrical and Computer Engineering, Rutgers, The State University of New Jersey, 94 Brett Road, Piscataway, NJ 08854, U.S.A.
†E-mail: parashar@caip.rutgers.edu

Contract/grant sponsor: National Science Foundation; contract/grant numbers: ACI 9984357 (CAREERS); EIA 0103674 (NGS); EIA-0120934 (ITR)
Contract/grant sponsor: Department of Energy/California Institute of Technology (ASCI); contract/grant number: PC295251

Received 28 December 2001
Revised 28 July 2002

simulations enables the experts to drive the discovery process by observing intermediate results, to change parameters to lead the simulation into more interesting domains, to detect and correct unstable situations, to dynamically inject updated information into the simulation, to play what-if games, and to terminate uninteresting runs early.

Enabling the seamless monitoring and interactive steering of parallel and distributed applications presents many challenges. A significant challenge is the definition and deployment of *sensors* and *actuators* [1,2] required to monitor and control application objects (algorithms and data structures). Defining these interaction interfaces and mechanisms in a generic manner and co-locating them with the application's computational objects can be non-trivial. This is because the structure of application computational objects vary significantly and the 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 the computational objects can be created, deleted, modified, migrated and redistributed on the fly. Another issue is the construction of a *control network* that interconnects these sensors and actuators so that commands and requests can be routed to the appropriate set(s) of computational objects (depending on current distribution of the object) and the returned information can be collated and coherently presented. Finally, the interaction and steering interfaces presented by the application need to be exported so that they can be accessed remotely, using standard distributed object protocols, to enable application monitoring and control.

This paper presents the design, implementation and experimental evaluation of DIOS (Distributed Interactive Object Substrate), an interactive object infrastructure that addresses these issues. The paper makes three key contributions.

1. Definition and deployment of interaction objects that encapsulate sensors and actuators for interrogation and control. Interaction objects can be distributed (spanning many processors) and dynamic (be created, deleted, migrated or distributed) and can be derived from existing computational data-structures. Traditional (C, Fortran, etc.) data-structures can be transformed into interaction objects using C++ wrappers.
2. Definition of a scalable control network interconnecting the interaction objects. The control network enables discovery, interaction and control of distributed computational objects and manages dynamic object creation, deletion, migration and redistribution. The control network is hierarchical and is designed to support applications executing on large parallel/distributed systems. An experimental evaluation of the control network is presented.
3. Definition of an Interaction Gateway that uses Java native interface (JNI) to provide a Java enabled proxy to the application for interaction and steering. The interaction gateway enables remote clients to connect to and access an application's computational objects (and thus its interaction interfaces) using standard distributed object protocols such as CORBA [3] and Java remote method invocation (RMI) [4].

DIOS has been implemented as a C++ library and is currently being used to enable interactive monitoring, steering and control of a wide range of scientific applications, including oil reservoir, compressible turbulence and numerical relativity simulations. It is a part of DISCOVER[‡], an ongoing

[‡]Information about the DISCOVER computational collaboratory can be found at http://www.caip.rutgers.edu/TASSL/Projects/DISCOVER/.

research initiative aimed at developing a Web-based interactive computational collaboratory. Its goal is to enable geographically distributed clients to collaboratively connect to, monitor and steer applications using Web-based portals. DIOS provides a high-level application programming interface (API) for generating interaction objects from existing computational objects, for registering these objects with DISCOVER and exporting their interaction interfaces, and for handling interaction/steering requests. The experimental evaluation of DIOS presented here demonstrates that overheads introduced by DIOS are small and insignificant when compared to application execution times.

The rest of the paper is organized as follows. Section 2 discusses related work in collaboration, interaction and steering. Section 3 presents an overview of the DISCOVER collaboratory. Section 4 presents the design of DIOS. It describes the creation, deployment and operation of interaction objects and the DIOS control network. Section 5 presents an experimental evaluation of DIOS. Section 6 presents a summary and concluding remarks.

2. RELATED WORK

Recent years have seen the development of a number of interactive computational problem-solving environments (PSEs) that address different aspects of application composition, configuration and execution. Similarly, a number of groupware infrastructures that provide collaboration capabilities have evolved. A classification and brief description of existing interactive and collaborative PSEs is presented below. Detailed surveys have been presented by Vetter *et al.* in [5] and van Liere *et al.* in [6]. The primary goal of DIOS/DISCOVER is to combine these two capabilities of interaction and collaboration to provide a collaborative PSE for application interaction and control.

1. *Interactive program construction.* Systems in this category, e.g. SCIRun [7], provide support for interactive program composition and construction. SCIRun allows users to graphically connect program components to create a data-flow style program graph. It is primarily targeted to developing new applications. While SCIRun does provide some steering capabilities, it does not support simultaneous steering of multiple applications or collaborative interaction and steering by multiple users.
2. *Performance optimization.* These systems are aimed at optimizing performance of applications. For example, in the Autopilot [8] system, sensors have a variety of sensor policies that optimize application performance. However, they do not directly support accessing objects at the application level for interactive monitoring and steering.
3. *Remote application configuration and deployment.* Systems in this category provide powerful visual authoring toolkits to configure and deploy distributed applications on existing high-performance metacomputing back-end resources. The CoG Kits [9] provide commodity access to the Globus [10] metacomputing environment. The WebFlow [11] and Gateway [12] systems provide support for configuring, deploying and analyzing distributed applications. These systems, however, do not provide any support for runtime application-level interaction and steering.
4. *Runtime interactive steering and control.* Systems in this category can be further classified based on the type and level of the interaction support provided.

(a) *Event-based steering systems.* In these systems, monitoring and steering actions are based on low-level system 'events' that occur during program execution. Application code is instrumented and interaction takes place when pre-defined events occur. The Progress [13] and Magellan [14] systems use this approach and require a server process executing in the same address space as the application to enable interaction. The computational steering environment (CSE) [15] uses a data manager as a blackboard for communicating data values between the application and the clients.

(b) *High-level abstractions for steering and control.* The Mirror Object Steering System (MOSS) [16,17] provides a high-level model for steering applications. Mirror objects are analogues to application objects (data structures) and are used for monitoring and steering. These objects export application methods to the interactivity system through which steering actions are accomplished. We believe that high-level abstractions for interaction and steering provide the most general approach for enabling interactive applications. The DIOS control network extends this approach.

5. *Collaboration groupware.* These environments include DOVE [18], the Web Based Collaborative Visualization [19] system, the Habanero [20] system, Tango [21], CCASEE [22] and CEV [23]. These systems primarily focus on enabling collaboration; some of them, however, do provide support for problem solving. The Tango system is based on a centralized server and is browser enabled. Habanero uses a Java-based centralized server architecture for Web-based collaboration (and collaborative visualization). CCASEE provides a distributed workspace using Java RMI. The CEV system provides collaborative visualization using a central server to perform the computations necessary to generate new collaborative views. The DOVE and the Web Based Collaborative Visualization systems also provide similar support for collaborative visualization.

The DISCOVER computational collaboratory brings together key technologies in interactive application frameworks, application interaction and steering mechanisms, interactive/collaborative Web Portals and servers, and collaboration groupware. DIOS provides high-level support for application monitoring and steering. It advances existing approaches in the following ways.

1. DIOS provides mechanisms for monitoring and interacting with application objects, which are both distributed (span multiple address spaces) and dynamic (can be dynamically created, deleted and migrated between address spaces). This allows any interactive computational objects to join or leave the control network on the fly. Existing interaction systems do not support distributed or dynamic objects.
2. DIOS defines a scalable control network to connect the distributed and dynamic interaction objects and their sensors and actuators, and provide external access to their interaction interfaces.
3. DIOS provides a Java-enabled application interaction proxy that allows remote access to the interaction objects and interfaces using standard distributed object technologies such as Java RMI and CORBA.

3. DISCOVER: AN INTERACTIVE COMPUTATIONAL COLLABORATORY

The DISCOVER [24,25] computational collaboratory provides a virtual, interactive and collaborative PSE that enables geographically distributed scientists and engineers to collaboratively monitor

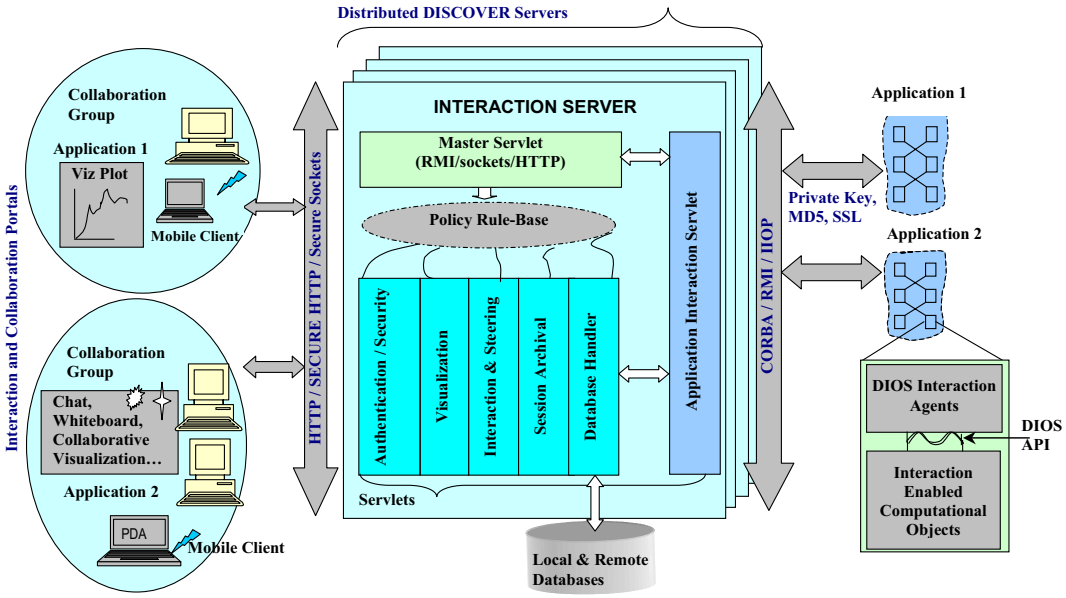


Figure 1. Architectural schematic of the DISCOVER computational collaboratory.

and control high-performance parallel/distributed applications. An architectural overview of the DISCOVER collaboratory is presented in Figure 1. Its front-end is composed of detachable client portals. Clients can connect to a DISCOVER server at any time using a browser to receive information about registered applications that are currently executing on remote systems. Furthermore, they can form or join collaboration groups and can (collaboratively) interact with one or more applications based on their capabilities. A network of interaction and collaboration servers forms the middle tier. These servers extend Web-servers with interaction and collaboration capabilities using Java Servlets [26]. The server architecture consists of a master acceptor/controller servlet and a suite of service handler servlets including interaction and steering handler, collaboration handler, security/authentication handler, visualization handler, session archival handler and database handler. Distributed servers are interconnected using CORBA and IIOP [3,27]. The back-end consists of applications enhanced with the DIOS control network and composed of interactive computational objects and distributed interaction agents.

The DISCOVER/DIOS interaction model is application initiated, i.e. the application registers with the server, exporting interaction interfaces composed of *views* and *commands* for different application objects. The interaction interfaces are defined and exported using high-level abstractions provided by DIOS. Views encapsulate sensors and provide information about the application and the application objects, while commands encapsulate actuators and process steering requests. Some or all of these

views/commands may be collaboratively accessed by groups of clients based on the clients' resources and capabilities. Session management and concurrency control is based on the capabilities provided by the middle (server) tier. A locking protocol is used to ensure that the application remains in a consistent state.

Security, client authentication and application access control is managed by a dedicated security and authentication handler at each server. In the current implementation, a two-level client authentication is performed on client login; the first level authorizes access to the server and the second level permits access to a particular application. On successful authentication at the server, the user is presented with a list of only those applications to which they have access privileges. A second level authorization is then performed for the chosen application to determine the access level (views only, views and commands, etc.) granted to the user and a customized access interface is created to match the authorized access level. To enable access control, all applications are required to provide a list of users and their access privileges. The application can also provide access privileges (typically view-only) to the 'world'. On the client side, digital certificates are used to validate the server identity before views are downloaded. A secure socket layer (SSL) provides encryption for all communication between the client and the server, while private key encryption and MD5 signatures are used to maintain the integrity of all communication between the application and server.

DISCOVER is currently operational (see http://www.discoverportal.org) and is being used to provide interaction capabilities to a number of scientific and engineering applications, including oil reservoir, computational fluid dynamics and numerical relativity simulations. The design, implementation and evaluation of DIOS are presented in the following sections.

4. DIOS

DIOS is composed of two key components: (1) interaction objects that extend computational objects with sensors and actuators; and (2) a hierarchical control network composed of *Discover Agents*, *Base Stations* and an *Interaction Gateway*, that interconnects interaction objects and provides access to them.

4.1. Sensors, actuators and interaction objects

Interaction objects extend application computational objects with interaction and steering capabilities, by providing them with co-located sensors and actuators. Computational objects are the objects (data-structures, algorithms) used by the application for its computations[§]. In order to enable application interaction and steering, these objects must export interaction interfaces that enable their state to be externally monitored and changed. Sensors and actuators provide such an interface. Sensors provide an interface for viewing the current state of the object, while actuators provide an interface for processing commands to modify the state. Note that sensors and actuators must be co-located in memory with the computational objects and have access to their internal state. Transforming computational objects into interaction objects can be a significant challenge. This is especially true when the computational objects are distributed across multiple processors and can be dynamically created, deleted, migrated

[§]Note that computational objects do not only refer to objects in an object-oriented implementation of an application, but also to application data structures and operations on these data-structures implemented in languages such as C and Fortran.

and redistributed. Multiple sensors and actuators now have to coordinate and collectively process interaction requests.

DIOS provides application-level programming abstractions and efficient runtime support to support the definition and deployment of sensors and actuators for distributed and dynamic computational objects. Using DIOS, existing applications can be made interactive by deriving the computational objects from a DIOS virtual interaction base class. The derived objects can then selectively overload the base class methods and define their interaction interfaces as a set of views that they can provide and a set of commands that they can accept and process. Views represent sensors and define the type of information that the object can provide. For example, a Grid object might export views for its structure and distribution. 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. This process requires minimal modification to original computational objects. Discover agents, part of the DIOS control network, combine the individual interfaces and export them to the interaction server using an Interaction interface definition language (IDL). The Interaction IDL contains metadata for interface discovery and access, and is compatible with standard distributed object interfaces like CORBA and RMI. In the case of applications written in non-object-oriented languages such as Fortran, application data structures are first converted into computation objects using a C++ wrapper object. Note that this only has to be done for those data structures that require interaction capabilities. These objects are then transformed into interaction objects as described above. DIOS interaction objects can be created or deleted during application execution and can migrate between computational nodes. Furthermore, a distributed interaction object can modify its distribution at any time.

4.2. Local, global and distributed objects

Interaction objects can be classified based on the address space(s) they span during the course of the computation as *local*, *global* and *distributed objects*. Local interaction objects are created in a processor's local memory. These objects may migrate to another processor during the lifetime of the application, but always exist in a single processor's address space at any time. Multiple instances of a local object may exist at the same time on different processors. Global interaction objects are similar to local objects, except that there can be exactly one instance of the object at any time, which is replicated on all processors. 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 its current distribution type (e.g. blocked, staggered, inverse space-filling curve-based or custom) and layout. This attribute can change during the lifetime of the object. Like local and global interaction objects, distributed objects can be dynamically created, deleted, or redistributed.

In order to enable interaction with distributed objects, each distribution type is associated with *gather* and *scatter* operations. Gather aggregates information from the distributed components of the object, while scatter performs the reverse operation. For example, in the case of a distributed array object, the gather operation would collate views generated from sub-blocks of the array while the scatter operator would scatter a query to the relevant sub-blocks. An application can select from a library of gather/scatter methods for popular distribution types provided by DIOS, or can register gather/scatter methods for customized distribution types.

4.3. Definition and deployment of interaction objects

Transforming an existing computational object into an interaction object using DIOS requires three steps as described below.

1. The computational object is derived from an appropriate virtual interaction class, depending on whether it is local, global or distributed.
2. Views and commands relevant to the computational object are defined. This involves defining methods that implement the desired functionality (generate a view or execute a command), if they do not already exist. For example, computing the desired one-dimensional slice corresponding to a *1-D Plot* view, or setting the value of a variable in response to a *SetValue* command.
3. Views and commands are registered with the DIOS control network. Registration consists of defining the tuple, `<Name, Callback, DataType>` for each view/command. *Name* is a unique name for the view/command that will be presented to the user. *Callback* is the method invoked to process the view or command. *DataType* is the type of the information returned by the view or command. It enables the front-end to invoke an appropriate viewer to present the returned information to the user. For example, a `TEXT` type indicates that the returned information is a simple text string, while a `1DPLOT` value indicates that it is a one-dimensional plot. Note that the virtual function `processMessage()` can be overridden to explicitly handle responses to views and command messages.

The process is illustrated in Figure 2. This figure shows a *GridFunction* computational object being derived from the appropriate DIOS base class to transform it into a local interactive object. Object initialization and definition of views and commands is performed in the object constructor which are not shown here for simplicity. In the figure, two views (*getGridDetails* and *getXYSlice*) and one command (*refineGrid*) are defined for the *GridFunction* object. Finally the *processMessage* method is overloaded to process interaction requests and invoke the appropriate callback functions.

As mentioned earlier, non-object-oriented (C/Fortran) data-structures can be converted into interaction objects by first defining C++ wrappers for these objects and then following the steps listed above. Although this requires some application modification, the wrappers are only required for those data-structures that have to be made interactive and the effort required is far less than rewriting the entire application to be interactive. We have successfully applied this technique to enable interactivity within the Fortran-based IPARS [28] parallel oil-reservoir simulator developed at the Center for Subsurface Modeling, University of Texas at Austin.

4.4. DIOS control network and interaction agents

The control network has a hierarchical ‘cellular’ structure with three components, Discover Agents, Base Stations and the Interaction Gateway, as shown in Figure 3. Computational nodes are partitioned into interaction cells, each cell consisting of a set of Discover Agents and a Base Station. The number of nodes per interaction cell is programmable. Discover Agents are present on each computational node and manage runtime references to the interaction objects on the node. The Base Station maintains information about interaction objects for the entire interaction cell. The highest level of the hierarchy is the Interaction Gateway which provides a proxy to the entire application. The control network is


```
/* Definition of the class GridFunction */
class GridFunction : public DIOS_LocalInteractionObject
{
    /* all data structures used for computation */
     public:
       // VIEWS
       char* getGridGetails();
       char* getXYSlice();

       // COMMANDS
       char* refineGrid(int refine_factor);
};

/* Implementation of the class GridFunction */
GridFunction::GridFunction() : DIOS_LocalInteractionObject()
{
    /* Exporting interaction information to DIOS */
    setObjectName("GridFunction");
    setObjectDescription("2-Dimensional Grid Function");
    setObjectType(DIOS_LOCAL_NON_DISTRIBUTED);

    /* Exporting interaction interfaces to DIOS */
    addView("getGridDetails", TEXT);
    addView("getXYSlice", 1DPLOT);
    addCommand("refineGrid", TEXT);
}

/* Overriding DIOS virtual function for run-time interaction */
GridFunction::processMessage(Message* msg)
{
    char* response;
    if(msg->msg_type() == VIEW_MESSAGE)
    {
         if(msg->interface() == "getGridDetails")
         {
       response = getGridDetails();
       setObjectResponse("getDetails", response);
        }
        if(msg->interface() == "getXYSlice")
        {
       response = getXYSlice();
       setObjectResponse("getXYSlice", response);
        }
    }
    else if (msg->msg_type() == COMMAND_MESSAGE)
    {
         if(msg->interface() == "refineGrid")
         {
       response = refineGrid(msg->params());
       setObjectResponse("refineGrid", response);
         }
    }
}
```

Figure 2. Definition and registration of an interaction object using DIOS.

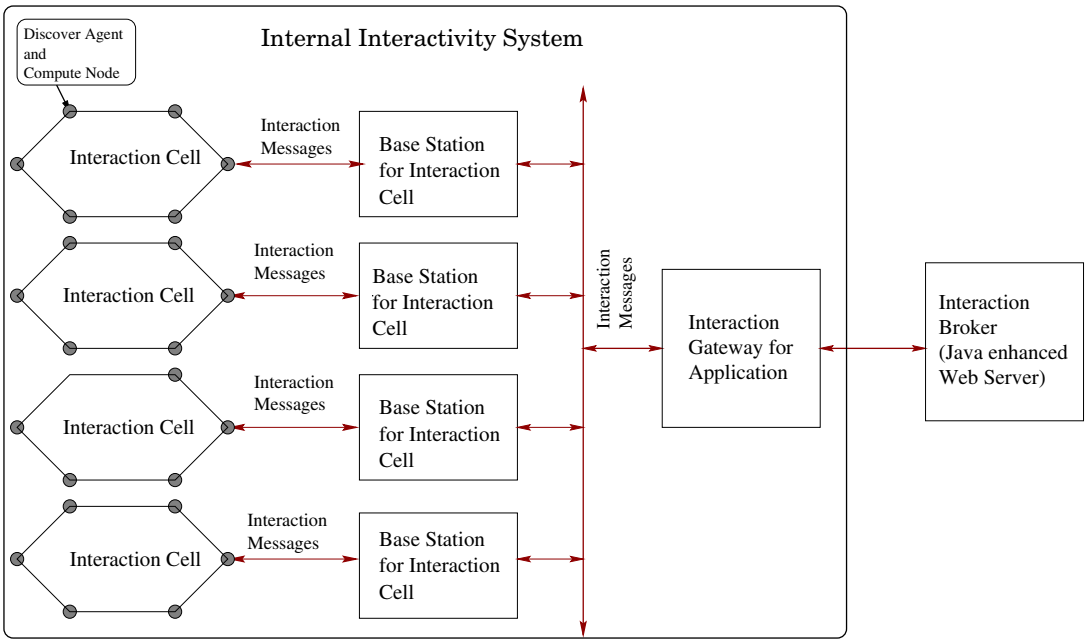


Figure 3. The DIOS control network.

automatically configured at runtime using the underlying messaging environment (e.g. message passing interface (MPI) [29]) and the available number of processors.

4.4.1. Discover Agents, Base Stations and Interaction Gateway

Each computational node in the control network houses a Discover Agent. Each Discover Agent maintains a local interaction object registry containing references to all interaction objects currently active and registered by that node, and exports the interaction interfaces for these objects (defined using the Interaction IDL). Base Stations form the next level of the control network hierarchy. They maintain interaction registries containing the Interaction IDL for all the interaction objects in the interaction cell, and export this information to the Interaction Gateway. The Interaction Gateway represents an interaction proxy for the entire application. It manages a registry of the interaction interfaces for all the interaction objects in the application and is responsible for interfacing with external interaction servers or brokers. The Interaction Gateway delegates incoming interaction requests to the appropriate Base Stations and Discover Agents and combines and collates responses. Object migrations and re-distributions are handled by the respective Discover Agents (and Base Stations if the migration/re-distribution is across interaction cells) by updating corresponding registries. The Discover Agent, Base Station and Interaction Gateway are all initialized on the appropriate processors during application start up. They execute in the same address space as the application and communicate using the application messaging environment, e.g. MPI.

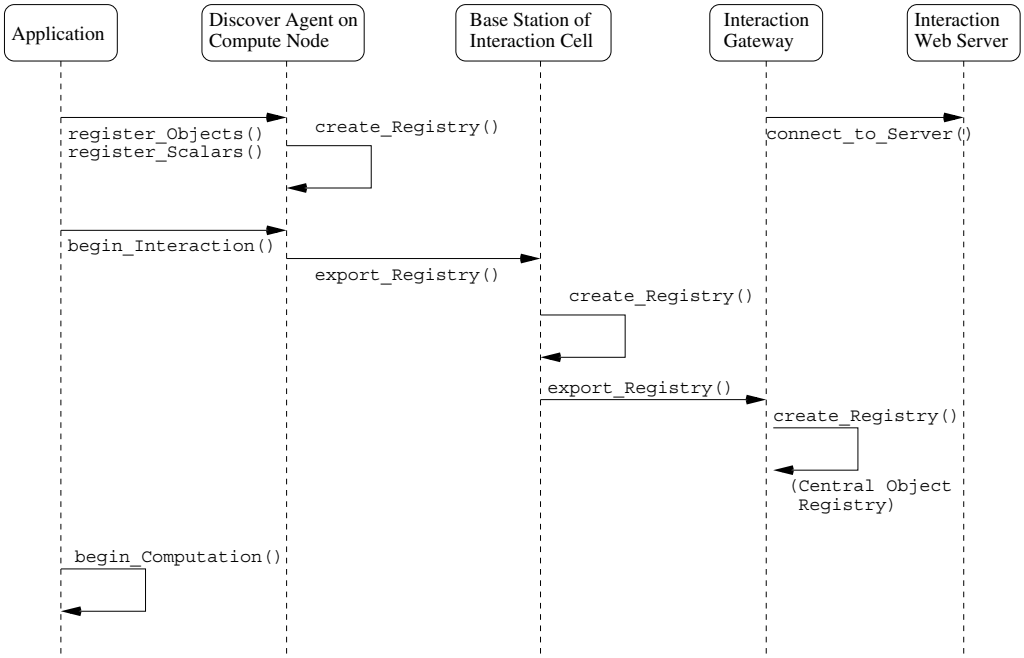


Figure 4. Initialization of the DIOS control network.

In our implementation, interactions between an interaction server and the interaction gateway are achieved using two approaches. In the first approach, the Interaction Gateway serializes the interaction interfaces and associated meta-data information for all registered interaction objects to the server. A set of Java classes at the server parse the serialized Interaction IDL stream to generate corresponding interaction object proxies. In the second approach, the Interaction Gateway initializes a Java Virtual Machine (JVM) and uses the JNI [30] to create Java mirrors of registered interaction objects. These mirrors are registered with a RMI [4] registry service executing at the Interaction Gateway. This enables the Server to gain access to and control the interaction objects using the Java RMI API. We are currently evaluating the performance overheads of using Java RMI and JNI. The use of the JVM and JNI in the second approach assumes that the computing environment supports the Java runtime environment.

A more detailed description of the DIOS framework, including examples for converting existing applications into interactive ones, registering them with the DISCOVER interaction server and using Web portals to monitor and control them, can be found in [31].

4.5. DIOS control network initialization and interaction sequences

The sequence of events during initialization of the DIOS control network is shown in Figure 4. During initialization, the application uses the DIOS API to create and register its interaction objects

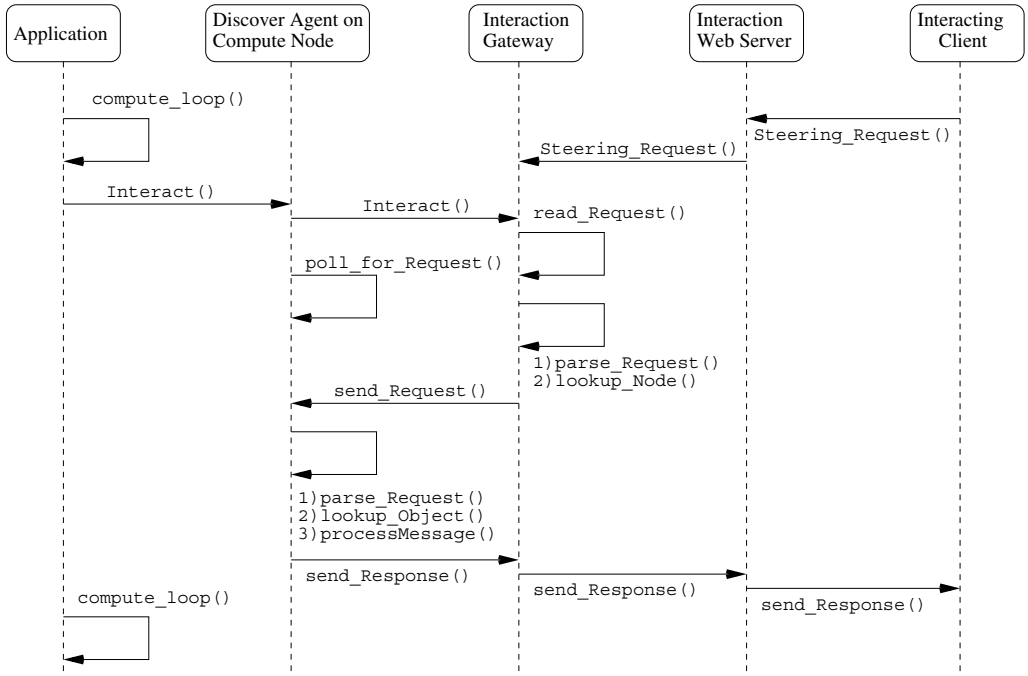


Figure 5. Interaction using the DIOS control network.

with the local Discover Agents. Interaction cells are then set up and the Base Stations establish communication with their respective Discover Agents to initialize their cell object registries. At the Interaction Gateway, the central interaction object registry is initialized. The Discover Agents export local object registries to their respective Base Stations, which in turn forward them to the Interaction Gateway. The Interaction Gateway now communicates with the DISCOVER server, registering the application and exporting the central object registry. At the server, the incoming Interaction IDL streams are parsed and interaction object proxies are recreated. Once the initial object registration process is complete, the application begins its computations.

The sequence of events during interaction is shown in Figure 5. During the interaction phase, the Interaction Gateway looks for any outstanding interaction requests from the server. If requests exist, it parses the request headers to identify the compute node(s) where the corresponding object resides and forwards the interaction request to the node(s). All other nodes are sent a go-ahead message indicating that there is no interaction request for any of the objects registered at these nodes. The Interaction Gateway then waits until the corresponding response arrives from the Discover Agent(s). If the responding object is distributed, the Interaction Gateway performs a gather operation on the individual responses. The response is then shipped to the server.

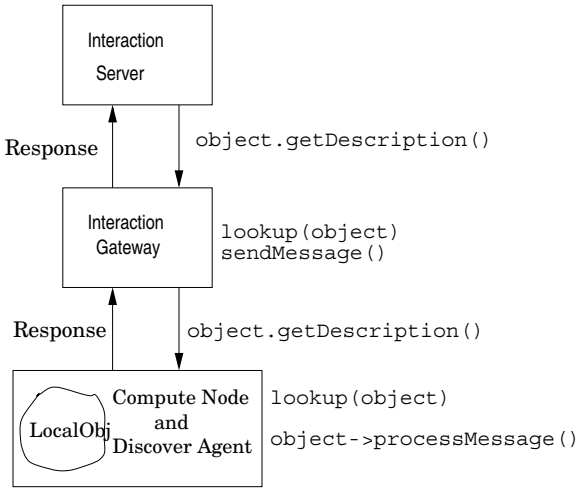


Figure 6. Interaction with a local object.

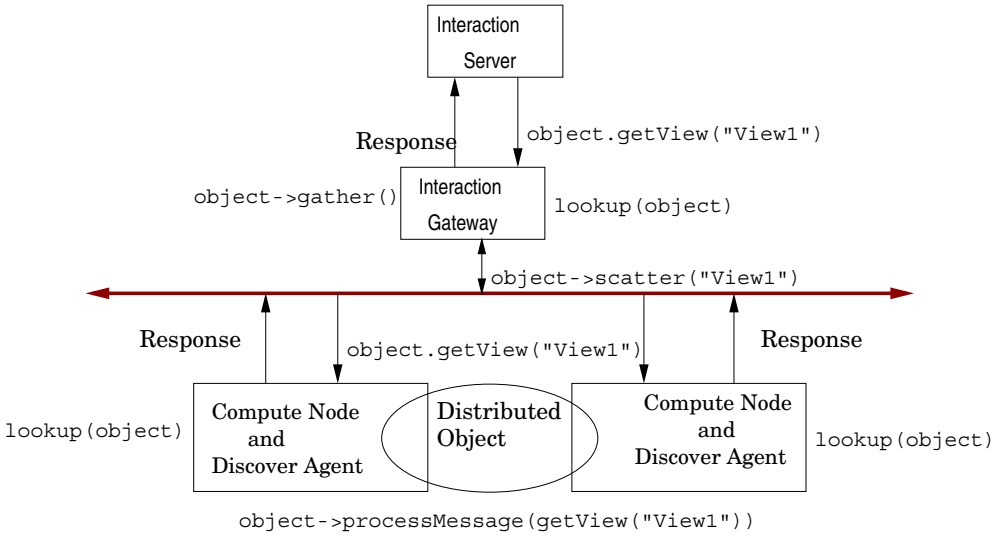


Figure 7. Interaction with a distributed object located on two remote nodes.

4.6. Interacting with local and distributed objects

The processing of interaction requests is slightly different for local and distributed objects. In the case of a local object residing on a single computational node, processing is straightforward and is shown in Figure 6. On receiving the request from the server, the Interaction Gateway parses the message header and looks into its registry to identify the computational node where the object resides. The request is then forwarded to the appropriate node. The corresponding Discover Agent on the node uses a reference to the associated interaction object to process the request. The response generated is then sent back to the Interaction Gateway, which in turn exports it to the server.

In the case of a distributed object (see Figure 7), the Interaction Gateway parses the message header to identify the object's distribution type. The corresponding *scatter* function is invoked to forward the request to the nodes across which the object is distributed. The corresponding Discover Agents receive the request, look up the associated interaction object and locally process the message. Each Discover Agent sends its portion of the response back to the Interaction Gateway, which then performs a *gather* operation to collate the responses and forwards them to the server.

5. EXPERIMENTAL EVALUATION

DIOS has been implemented as a C++ library and has been ported to a number of operating systems including Linux, Windows NT, Solaris, IRIX and AIX. This section summarizes an experimental evaluation of the DIOS library using the IPARS reservoir simulator framework on the Sun Starfire E10000 cluster. The E10000 configuration used consisted of 64, 400 MHz SPARC processors and a 12.8 Gbytes/second interconnect.

The evaluation of DIOS consisted of five experiments: (1) measurement of overheads during initialization and object registration; (2) measurement of runtime overheads during steering; (3) measurements of view/command processing times; (4) measurement of overheads due to the control network; and (5) measurement of end-to-end steering latency.

5.1. Integrated Parallel Accurate Reservoir Simulator

Integrated Parallel Accurate Reservoir Simulator (IPARS) is a Fortran-based framework for developing distributed reservoir simulators. It provides a high-level interface for reservoir specification and composition and a runtime system that supports data distribution and communications. The primary motivation for transforming IPARS into an interactive application was to enable scientists and petroleum engineers in the field to be able to use simulation results to drive the drilling of wells and optimize oil production. Using DIOS/DISCOVER they can interactively feed in parameters such as water/gas injection rates and well bottom hole pressure, and observe the water/oil ratio or the oil production rate. The transformation of IPARS using DIOS consisted of creating C++ wrappers around the IPARS well data structures and defining the appropriate interaction interfaces in terms of views and commands. Details about the IPARS–DIOS–DISCOVER integration can be found at http://www.ticam.utexas.edu/CSM/ACTI/ipars.html. A screen-dump of the DISCOVER–IPARS portal during an interaction session is shown in Figure 8.

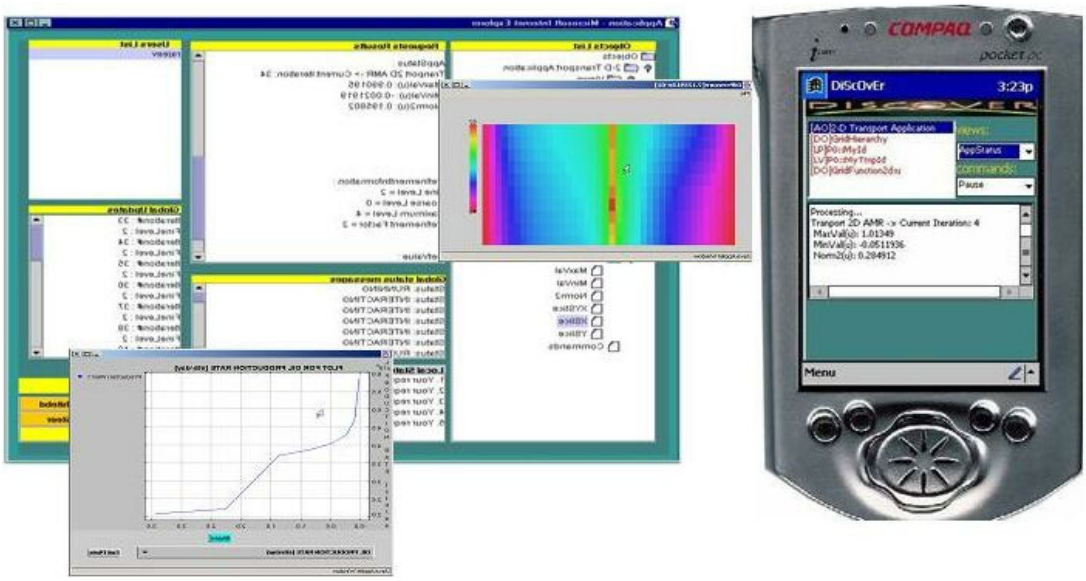


Figure 8. The DISCOVER portal for interaction and steering.

5.2. Interaction object registration

In this experiment we measured the time required for object registration during initialization. This includes generating the Interaction IDL for each interaction object at the Discover Agents and exporting this IDL to the Base Station and the Interaction Gateway. These average measured overheads were 500 μs per object at each Discover Agent, 10 ms per Discover Agent in the interaction cell at the Base Station and 10 ms per Base Station in the control network at the Gateway. Object initialization and registration is the most expensive DIOS operation. We are currently working on optimizing the registration process to reduce this overhead. This cost, however, is a one-time setup cost and is only required during initialization.

5.3. Overhead of minimal steering

In this set of experiments we measured the runtime overheads introduced by DIOS during application execution. In this experiment, the application executed in the minimal steering mode, i.e. the application automatically updated the DISCOVER server and connected clients with the current state of its interactive objects. Explicit command/view requests (other than those automatically provided by the application) were disabled during the experiment. The application contained five interactive objects, two local and three global. The application's runtimes with and without DIOS are plotted in

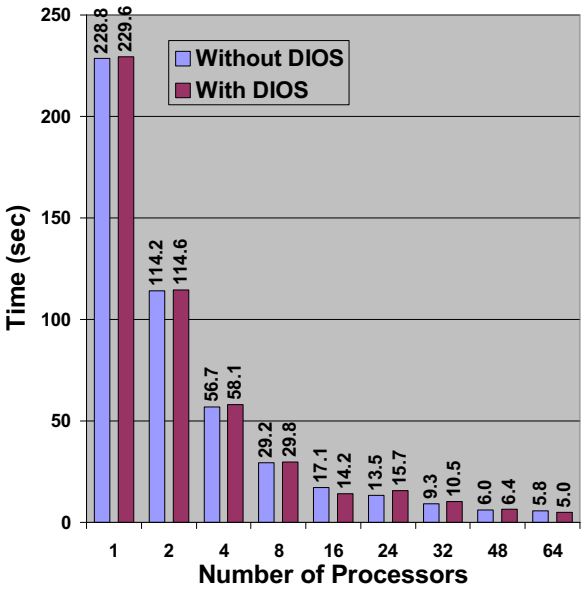


Figure 9. Overhead due to DIOS runtime monitoring in the minimal steering mode.

Figure 9. It can be seen that the overheads due to the DIOS runtime are very small and typically within the error of measurement. In some cases, due to system load dynamics, the performance with DIOS was slightly better. Our observations have shown that, for most applications, the DIOS overheads are less than 0.2% of the application computation time.

5.4. View/command processing time

The query processing and steering time depends on the nature of interaction/steering requested, the processing required at the application to satisfy the request and generate a response, and the type and size of the response. In this experiment we measured the time required for generating and exporting views and for processing commands. The time for generating and exporting views for return data sizes ranging from a few bytes to 1 kbyte was between 1 and 2 ms. Similarly, command-processing times depended on the nature of the command. Commands such as stop, pause and continue required only about 250 μs to process, while checkpoint and rollback commands requiring file I/O took longer. Note that, in this experiment, all the collaborating clients generated the same view and command requests. A sampling of these times for different view generations and command executions is presented in Table I.

Table I. View and command processing times.

View type	Data size (bytes)	Time taken (ms)	Command	Time taken
Text	65	0.4	Stop, Pause or Resume	250 μs
Text	120	0.7	Refine GridHierarchy	32 ms
Text	760	0.7	Checkpoint	1.2 s
XSlice Generation	1024	1.7	Rollback	43 ms

5.5. DIOS control network overheads

In this set of experiments we measured the overheads due to communication between the Discover Agents, Base Stations and the Interaction Gateway while processing interaction requests for local, global and distributed objects. As expected, the measurements indicated that the interaction request processing time is minimum when the interaction objects are co-located with the Gateway and is maximum for distributed objects. This is due to the additional communications between the different Discover Agents and the Gateway and the `gather` operation performed at the Gateway to collate the responses. Note that for the IPARS application, the average interaction time (for either local, remote or distributed objects) was within 0.1% to 0.3% of the average time spent in computation during each iteration. Figure 10 compares computation time with interaction times at the Discover Agent, Base Station and Interaction Gateway for successive application iterations. Note that the interaction times include the request processing times in addition to control network overheads. Figure 11 shows the breakdown of the interaction time. The control network overhead is the time spent interacting with other nodes and is significantly smaller than the request-processing time (local processing + message parsing). Finally, Figure 12 shows the breakdown of the interaction time in the case of an object distributed across three nodes. The interaction times are measured at the Interaction Gateway in this case.

5.6. End-to-end steering latency

In this set of experiments we measured the time to complete a round-trip steering operation starting with a request from a remote client and ending with the response delivered to that client. Remote clients executed within Web-browsers on laptops/workstations on different subnets. These measurements of course depend on the state of the client, the server and the network interconnecting them. The DISCOVER system exhibits end-to-end latencies between 10 and 45 ms for data sizes ranging from a few bytes to 10 kbytes, which is comparable to latencies for the MOSS and Autopilot steering systems as reported by Eisenhauer in [17]—these systems however do not support interactions with distributed or dynamic objects. Latencies for the DISCOVER, MOSS and Autopilot are plotted in Figure 13.

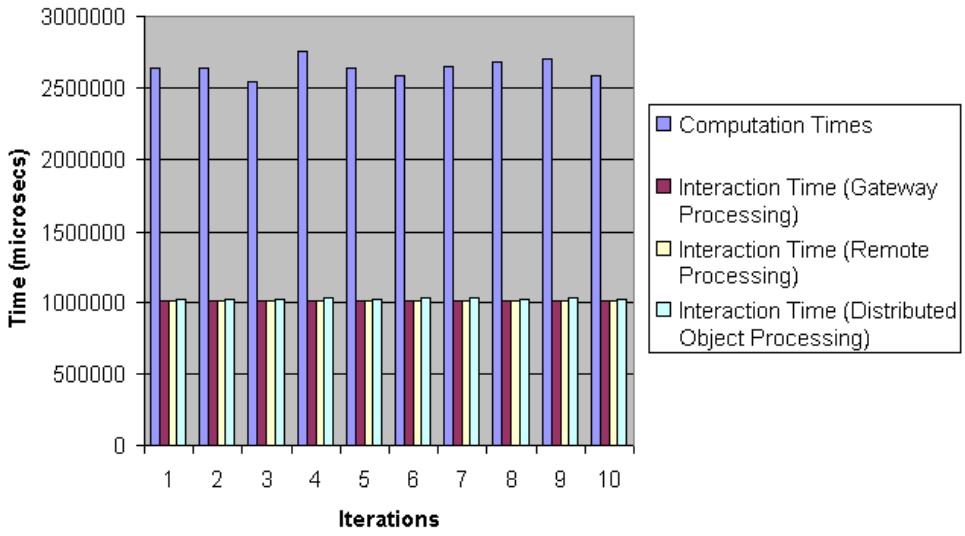


Figure 10. Comparison of computation and interaction times at each Discover Agent, Base Station and Interaction Gateway for successive application iterations.

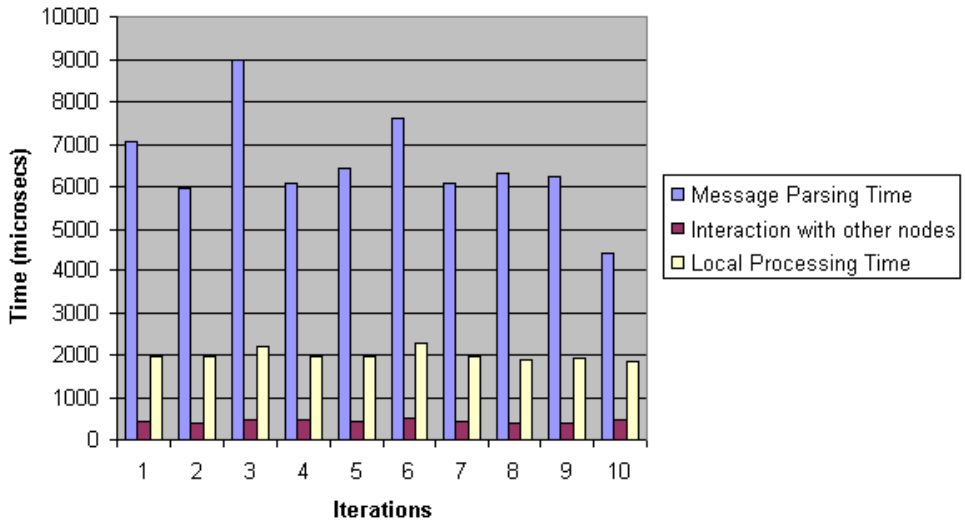


Figure 11. Breakdown of interaction overheads at the Interaction Gateway during successive application iterations.

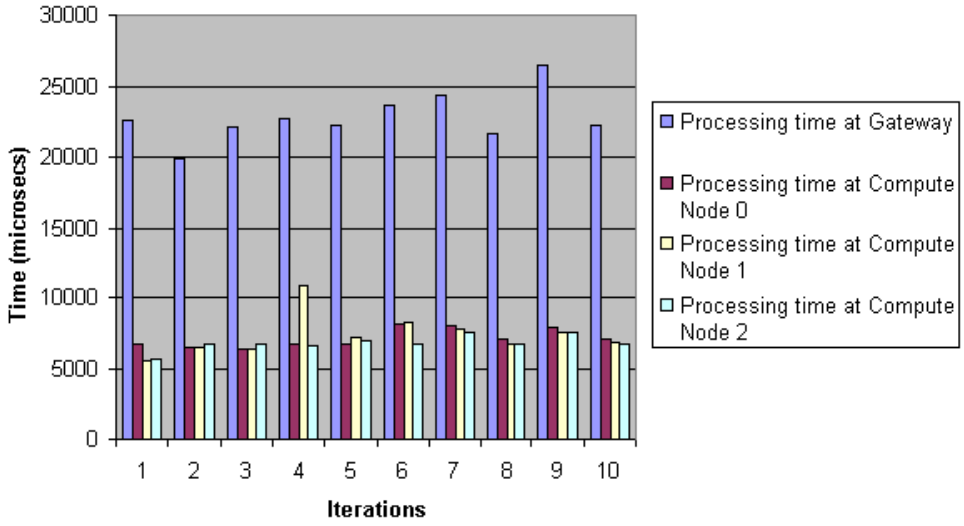


Figure 12. Breakdown of request-processing overheads for an object distributed across three compute nodes. The interaction time is measured at the Interaction Gateway.

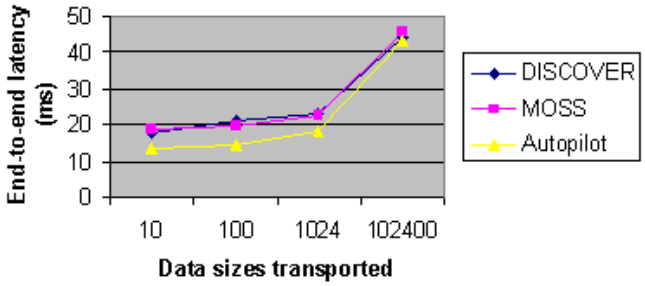


Figure 13. Comparison of end-to-end steering latencies of the DISCOVER, MOSS and Autopilot systems.

6. SUMMARY AND CONCLUSIONS

This paper presented the design, implementation and experimental evaluation of DIOS, an interactive object infrastructure to enable the runtime monitoring and computational steering of parallel and distributed applications. DIOS enables the high-level definition and deployment of sensors and actuators into existing application objects. Application objects may be distributed (spanning many processors) and dynamic (be created, deleted, changed or migrated). Furthermore, DIOS provides a control network that interconnects the interactive objects in a parallel/distributed application and

enables the external discovery, interrogation, monitoring and manipulation of these objects at runtime. The control network hierarchically interconnects object sensors and actuators and is designed to ensure scalability on large parallel and distributed systems. The Interaction Gateway provides a (Java-enabled) interaction 'proxy' to the application and enables remote access using standard distributed object protocols and Web browsers. An experimental evaluation of the DIOS framework was presented and demonstrated that overheads introduced by DIOS are small and insignificant when compared to application execution times. To further reduce the end-to-end application response latency, a multithreaded runtime system is being developed. DIOS is currently operational and is being used to provide interaction and steering capabilities to a number of application specific PSEs including the IPARS oil-reservoir simulator system at the Center for Subsurface Modeling, University of Texas at Austin, the Virtual Test Facility at the ASCI/ASAP Center, California Institute of Technology and the Astrophysical Simulation Collaboratory.

ACKNOWLEDGEMENTS

We would like to thank the members of the DISCOVER team, V. Bhat, M. Dhillon, S. Kaur, H. Liu, V. Mann, V. Matossian, A. Swaminathan and S. Verma for their contributions to this project. We would also like to thank W. Lee and J. Wheeler and M. Wheeler for their support in integrating IPARS with DISCOVER/DIOS. The research presented in this paper was supported in part by the National Science Foundation via grants numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS) and EIA-0120934 (ITR), and by Department of Energy/California Institute of Technology (ASCI) via grant number PC295251.

REFERENCES

1. Plale B, Eisenhauer G, Schwan K, Heiner J, Martin V, Vetter J. From interactive applications to distributed laboratories. *IEEE Concurrency* 1998; **6**(2):78–90.
2. Vetter J, Schwan K. High performance computational steering of physical simulations. *Proceedings 11th International Parallel Processing Symposium (IPPS '97)*, Geneva, Switzerland, 1–5 April 1997. IEEE Computer Society Press: Piscataway, NJ, 1997; 128–132.
3. CORBA: Common Object Request Broker Architecture. http://www.omg.org [17 July 2002].
4. Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi [17 July 2002].
5. Gu W, Vetter J, Schwan K. Computational steering annotated bibliography. *Sigplan Notices* 1997; **32**(6):40–44.
6. Mulder J, van Wijk J, van Liere R. A survey for computational steering environments. *Future Generation Computer Systems* 1999; **15**(1):119–129.
7. Parker SG, Johnson CR. The SCIRun computational steering software system. *Modern Software Tools in Scientific Computing*, Arge E, Bruaset AM, Langtangen HP (eds.). Birkhauser Press, 1997.
8. Ribler RL, Vetter JS, Simitci H, Reed DA. Autopilot: Adaptive control of distributed applications. *Proceedings 7th IEEE Symposium on High-Performance Distributed Computing*, 1998; 172–179.
9. von Laszewski G, Foster I, Gawor J. CoG Kits: A bridge between commodity distributed computing and high performance grids. *Proceedings of ACM 2000 Java Grande Conference*, San Francisco, CA, 2000. ACM Press: New York, 2002; 97–106.
10. Foster I, Kesselman C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications* 1997; **11**(2):115–128.
11. Bhatia D, Burzevski V, Camuseva M, Fox G, Furmanski W, Premchandran G. WebFlow—a visual programming paradigm for Web/Java based coarse grain distributed computing. *Concurrency—Practice and Experience* 1997; **9**(6):555–557.
12. Haupt T, Akarsu E, Fox G, Youn C-H. The Gateway system: Uniform Web based access to remote resources. *Concurrency—Practice and Experience* 2000; **12**(8):629–642.
13. Vetter J, Schwan K. Progress: A toolkit for interactive program steering. *Proceedings 24th International Conference on Parallel Processing*, vol. II, Urbana, IL, 1995. IEEE Computer Society Press: Piscataway, NJ, 1995; 139–142.
14. Vetter J, Schwan K. Models for computational steering. *Proceedings IEEE 3rd International Conference on Configurable Distributed Systems (ICCDS '96)*, Annapolis, MD, 1996. IEEE Computer Society Press: Piscataway, NJ, 1996; 42–52.

15. van Liere R, Harkes J, de Leeuw W. A distributed blackboard architecture for interactive data visualization. *Proceedings IEEE Visualization'98 Conference*, Research Triangle Park, NC, 1998. IEEE Computer Society Press: Piscataway, NJ, 1998; 225–231.
16. Eisenhauer G, Schwan K. An object-based infrastructure for program monitoring and steering. *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, Welches, OR, 1998. ACM Press: New York, 1998; 10–20.
17. Eisenhauer G. An object infrastructure for high-performance interactive applications. *PhD Thesis*, Department of Computer Science, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332, 1998.
18. Jain LK. A distributed, component-based solution for scientific information management. *MS Report*, Oregon State University, 1998.
19. Bajaj C, Cutchin S. Web based collaborative visualization of distributed and parallel simulation. *IEEE Parallel Symposium on Visualization*, 1999; 47–54.
20. Driggers B, Alameda J, Bishop K. Distributed collaboration for engineering and scientific applications implemented in Habanero, a Java-based environment. *Concurrency—Practice and Experience* 1997; **9**(11):1269–1277.
21. Beca L, Cheng G, Fox G, Jurga T, Olszewski K, Podgorny M, Sokolowski P, Stachowiak T, Walczak K. Tango—a collaborative environment for the World Wide Web. *Technical Report*, Northeast Parallel Architectures Center, Syracuse University, Syracuse NY, 1997.
22. Raje R, Teal A, Coulson J, Yao S, Winn W, Guy E III. CCASEE—A Collaborative Computer Assisted Software Engineering Environment. *Proceedings of the International Association of Science and Technology for Development (IASTED), SE'97 Conference*, San Francisco, CA, 1997; 367–371.
23. Boyles M, Raje RR, Fang S. CEV: Collaboration Environment for Visualization using Java-RMI. *Concurrency—Practice and Experience* 1998; **10**(11–13):1079–1085.
24. Muralidhar R, Kaur S, Parashar M. An architecture for Web-based interaction and steering of adaptive parallel/distributed applications. *Proceedings of the 6th International Euro-Par Conference (Euro-Par 2000) Proceedings of Euro-Par 2000* (*Lecture Notes in Computer Science*, vol. 1900). Springer: Berlin, 2000; 1332–1339.
25. Mann V, Matossian V, Muralidhar R, Parashar M. DISCOVER: An environment for Web-based interaction and steering of high-performance scientific applications. *Concurrency and Computation: Practice and Experience* 2001; **13**(8–9): 737–754.
26. Hunter J, Crawford W. *Java Servlet Programming* (2nd edn), 2001. O'Reilly and Associates: Sebastopol, CA, 2001.
27. CORBA/IIOP Specification. http://www.omg.org/technology/documents/formal/corba_iiop.htm [17 July 2002].
28. Wheeler JA *et al. IPARS (Integrated Parallel Accurate Reservoir Simulator) Center for Subsurface Modeling*, Texas Institute for Computational and Applied Mathematics, University of Texas at Austin. http://www.ticam.utexas.edu/CSM/ACTI/ipars.html [17 July 2002].
29. MPI Forum. MPI: Message Passing Interface. http://www-unix.mcs.anl.gov/mpi/ [17 July 2002].
30. Java Native Interface Specification. http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html [17 July 2002].
31. Muralidhar R. A distributed object framework for the interactive steering of high-performance applications. *MS Thesis*, Department of Electrical and Computer Engineering, Rutgers, The State University of New Jersey, 2000. http://www.caip.rutgers.edu/TASSL/Thesis/rajeev-thesis.ps [17 July 2002].

