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
{rajeevdm,parashar}@caip.rutgers.edu

Abstract. This paper presents the design, implementation and experimental evaluation of DIOS, an infrastructure for enabling the runtime monitoring and computational steering of parallel and distributed applications. DIOS enables existing application objects (data structures) to be enhanced with sensors and actuators so that they can be interrogated and controlled at runtime. Application objects can be distributed (spanning many processors) and dynamic (be created, deleted, changed or migrated). Furthermore, DIOS provides a control network that manages the distributed sensors and actuators and enables external discovery, interrogation, monitoring and manipulation of these objects at runtime. DIOS is currently being used to enable interactive monitoring and steering of a wide range of scientific applications, including oil reservoir, compressible turbulence and numerical relativity simulations.

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 the scientists and engineers to be able to monitor the progress of these simulations, and to control and steer them at runtime. Enabling seamless monitoring and interactive steering of parallel and distributed applications however, presents many challenges. A significant challenge is the definition and deployment of *sensors* and *actuators* 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 varies 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*

* Research supported by the National Science Foundation via grants number ACI 9984357 (CAREERS) awarded to Manish Parashar.

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 67–75, 2001.

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, to enable application monitoring and control.

This paper presents the design and evaluations of DIOS (Distributed Interactive Object Substrate), an interactive object infrastructure that supports application interaction and steering. DIOS addresses three key challenges - (1) Definition and deployment of interaction objects that encapsulate sensors and actuators for interrogation and control. Interaction objects can be distributed and dynamic, and can be derived from existing computational data-structures. Traditional (C, Fortran, etc.) data-structures can also be transformed into interaction objects using C++ wrappers. (2) Definition of a scalable control network interconnecting the interaction objects that 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 JNI (Java Native Interface) 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 and Java RMI.

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. DIOS is a part of DISCOVER [3][1], an ongoing project aimed at developing an interactive computational collaboratory that enables geographically distributed clients to collaboratively connect to, monitor and steer applications using web-based portals.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 presents the design and operation of DIOS. Section 4 presents an experimental evaluation of DIOS. Section 5 presents concluding remarks.

2 Related Work

This section briefly describes related work in computational interaction and steering. A detailed classification of existing interactive and collaborative PSEs (Problem Solving Environments) is presented in [5]. Other surveys have been presented by Vetter et. al in [2] and van Liere et. al in [4]. Run-time interactive steering and control systems can be divided into two classes based on the type and level of the interaction support provided. (1) *Event based steering systems:*

[1] Information about the DISCOVER project can be found at *http://www.caip.rutgers.edu/TASSL/Projects/DISCOVER/.*

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 [6] and Magellan [7] systems use this approach. (2) *High-level abstractions for steering and control:* The Mirror Object Steering System (MOSS) ([1]) provides a high-level model for steering applications. Mirror objects are analogues of application objects (data structures) and export application methods for interaction. We believe that high-level abstractions for interaction and steering provide the most general approach for enabling interactive applications. DIOS extends this approach.

3 DIOS: Distributed Interactive Object Substrate

DIOS is composed of 2 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 the interaction objects and provides access to them.

3.1 Sensors, Actuators and Interaction Objects

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 provide an interface for viewing the current state of the object, while actuators provide an interface to process commands to modify the state. Note that the sensors and actuators need to be co-located in memory with the computational objects and have access their internal state. If the computational objects are distributed across multiple processors and can be dynamically created, deleted, migrated and redistributed on-the-fly, multiple sensors and actuators now have to coordinate and collectively process interaction requests.

DIOS provides an API to enable applications to define sensors and actuators. This is achieved by simply deriving the computational objects from a virtual interaction base class provided by DIOS. The derived objects can then selectively overload the base class methods to define their interaction interface as a set of views (sensors) that they can provide and a set of commands (actuators) that they can accept. For example, a Grid object might export views for its structure and distribution. Commands for the Grid object may include refine, coarsen, and redistribute. This process requires minimal modification to original computational objects. Interaction interfaces are exported to the interaction server using a simple Interaction IDL (Interface Definition Language). 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. These objects are then transformed into interaction objects.

Interaction objects can be *local*, *global* or *distributed* depending on the address space(s) they span during the course of the computation. Local objects belong to a single address space and there could be multiple instances of a local object on different processors. They can also migrate to another processor at run time. Global objects are similar, but have exactly one instance (that could be 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 (blocked, staggered, inverse space filling curve-based, or custom) and layout. This attribute can change during the lifetime of the object, e.g. when the object is redistributed. 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.

3.2 DIOS Control Network and Interaction Agents

The control network has a hierarchical structure composed of three kinds of interaction agents, Discover Agents, Base Stations, and Interaction Gateway, as shown in Figure 1. 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. The control network is automatically configured at run-time using the underlying messaging environment (e.g. MPI) and the available number of processors.

Each compute node in the control network houses a Discover Agent that maintains a local interaction object registry containing references to all interaction objects currently active and registered by that node. At the next level of hierarchy, Base Stations maintain registries containing the Interaction IDL for all the interaction objects in an interaction cell. The Interaction Gateway represents an interaction proxy for the entire application and 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. During initialization, the application uses the DIOS API to create and register its interaction objects with local Discover Agents. The Discover Agents export the interaction IDL's for all these objects to their respective Base Stations. Base Stations populate their registries and then forward the interaction IDL's to the Interaction Gateway. The Interaction Gateway, after updating its registry communicates with the DISCOVER server, registering the application and exporting all registered objects. The application now begins its computations. The interaction between the Interaction Gateway and the DISCOVER server is managed by initializing a Java Virtual Machine and using the Java Native Interface to

create Java mirrors of all registered interaction objects. These mirrors are registered with a RMI (Remote Method Invocation) 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.

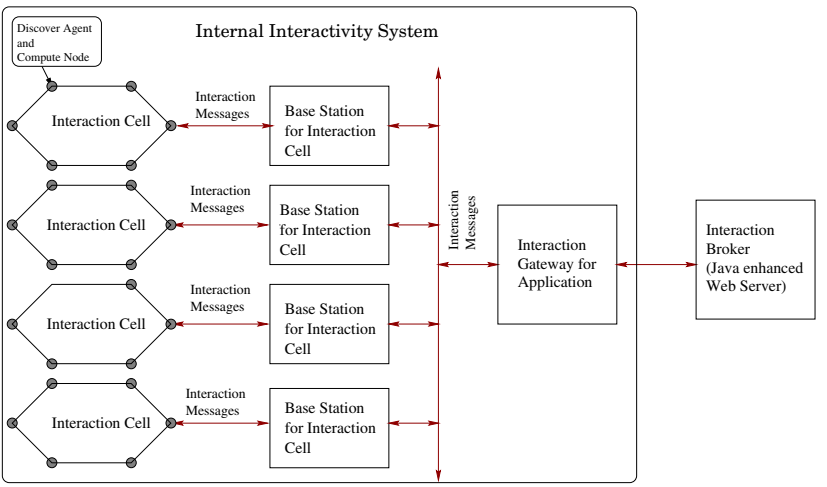

Fig. 1. The DIOS Control Network

During interaction phases, the Interaction Gateway delegates incoming interaction requests to the appropriate Base Stations and Discover Agents, and combines and collates responses (for distributed objects). 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. 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 the operation of the control network can be found in [5].

4 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 consists of 64, 400 MHz SPARC processors, a 12.8 GBytes/sec interconnect. IPARS is a Fortran-based framework for developing parallel/distributed reservoir simulators. Using DIOS/DISCOVER, engineers 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. The DIOS evaluation consists of 5 experiments:

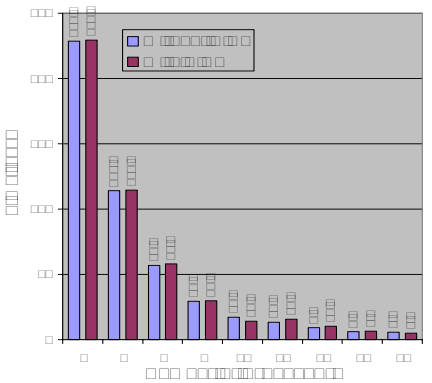

Fig. 2. Overhead due to DIOS runtime monitoring in the minimal steering mode

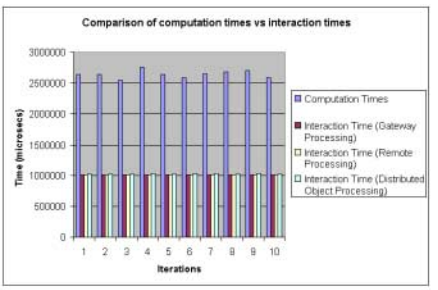

Fig. 3. Comparison of computation and interaction times at each Discover Agent, Base Station and Interaction Gateway for successive application iterations

Interaction Object Registration: Object registration (generating the Interaction IDL at the Discover Agents and exporting it to Base Station/Gateway) took 500 μsec 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. Note that this is a one-time cost.

Overhead of Minimal Steering: This experiment measured the runtime overheads introduced due to DIOS monitoring during application execution. In this experiment, the application automatically updated the DISCOVER server and connected clients with the current state of its interactive objects. Explicit command/view requests were disabled during the experiment. The application contained 5 interaction objects, 2 local objects and 3 global objects. The application's run times with and without DIOS are plotted in Figure 2. 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 that 0.2% of the application computation time.

View/Command Processing Time: The query processing time depends on - (a) the nature of interaction/steering requested, (b) the processing required at the application to satisfy the request and generate a response, and (c) type and size of the response. In this experiment we measured time required for generating

and exporting different views and commands. A sampling of the measured times for different scenarios is presented in Table 1.

Table 1. View and command processing times

View Type	Data Size (Bytes)	Time Taken	Command	Time Taken
Text	65	1.4 ms	Stop, Pause or Resume	250 μsec
Text	120	0.7 ms	Refine GridHierarchy	32 ms
Text	760	0.7 ms	Checkpoint	1.2 sec
XSlice Generation	1024	1.7 ms	Rollback	43 ms

DIOS Control Network Overheads: This experiment consisted of measuring 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 the maximum for distributed objects. This is due to the additional communication 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 was within 0.1 to 0.3% of the average time spent in computation during each iteration. Figure 3 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. Finally, Figure 4 shows the breakdown of the interaction time in the case of an object distributed across 3 nodes. The interaction times are measured at the Interaction Gateway in this case.

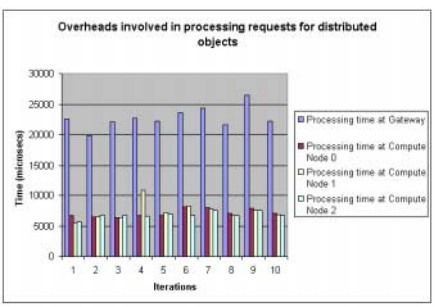

Fig. 4. Breakdown of request-processing overheads for an object distributed across 3 compute nodes. The interaction times is measured at the Interaction Gateway

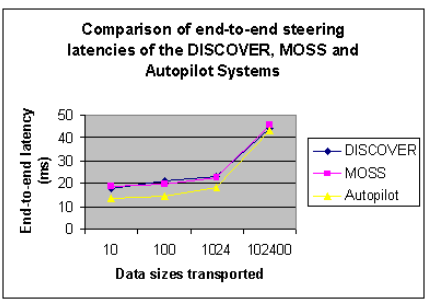

Fig. 5. Comparison of end-to-end steering latencies for DISCOVER, MOSS and Autopilot systems

End-to-end steering latency: This 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. 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 (shown in Figure 5) comparable to steering systems like the MOSS and Autopilot systems, as reported in [1].

5 Concluding Remarks

This paper presented the design, implementation and experimental evaluation of DIOS, an interactive object infrastructure for enabling the runtime monitoring and computational steering of parallel and distributed applications. The DIOS interactive object framework enables high-level definition and deployment of sensors and actuators into existing application objects. Furthermore, the DIOS control network and runtime system supports distributed and dynamic objects and can manage dynamic object creation, deletion, migration and redistribution. The Interaction Gateway provides an interaction proxy to the application and enables remote access using distributed object protocols and web browsers. An experimental evaluation of the DIOS framework was presented. DIOS is currently operational and is being used to provide interaction and steering capabilities to a number of application specific PSEs.

References

1. Eisenhauer, G.: *An Object Infrastructure for High-Performance Interactive Applications.* PhD thesis, Department of Computer Science, Georgia Institute of Technology, May 1998. 69, 74
2. Gu, W., Vetter, J., Schwan, K.: *Computational steering annotated bibliography.* Sigplan notices, 32 (6): 40-4 (June 1997). 68
3. Mann, V., Matossian, V., Muralidhar, R., Parashar, M.: *DISCOVER: An Environment for Web-based Interaction and Steering of High-Performance Scientific Applications.* To appear in Concurrency and Computation: Practice and Experience, John Wiley Publishers, 2001. 68
4. Mulder, J., van Wijk, J., van Liere, R.: *A Survey of Computational Steering Environments.* Future Generation Computer Systems, Vol. 15, nr. 2, 1999. 68
5. 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, October 2000. 68, 71
6. Vetter, J., Schwan, K.: *Progress: A Toolkit for Interactive Program Steering.* Proceedings of the 1995 International Conference on Parallel Processing, pp. 139-149. 1995. 69
7. Vetter, J., Schwan, K.: *Models for Computational Steering.* Third International Conference on Configurable Distributed Systems, IEEE, May 1996. 69

8. Wheeler, J., et al: *IPARS: Integrated Parallel Reservoir Simulator*. Center for Subsurface Modeling, University of Texas at Austin.
http://www.ticam.utexas.edu/CSM.