InterAct: Virtual Sharing for Interactive Client-Server Applications ⋆

Srinivasan Parthasarathy and Sandhya Dwarkadas

Department of Computer Science
University of Rochester
Rochester, NY 14627–0226

{srini,sandhya}@cs.rochester.edu

Abstract. *We describe InterAct, a framework for interactive client-server applications. InterAct provides an efficient mechanism to support object sharing while facilitating client-controlled consistency. Advantages are two-fold: the ability to cache relevant data on the client to help support interactivity, and the ability to extend the computation boundary to the client in order to reduce server load. We examine its performance on the interactive data-mining domain, and present some basic results that indicate the flexibility and performance achievable.*

1 Introduction

Many applications require interaction among disparate components, often running in different environments. Simulation of interactive virtual environments, interactive speech recognition, interactive vision (object recognition systems), and interactive data mining, are some examples. A large number of such applications are client-server in nature and involve processes that exchange data in an irregular fashion.

In such application domains, both the data sharing mechanisms and the consistency models are important. While the use of shared memory improves ease-of-use, employing a spatial memory model such as release consistency [6] is not always the most efficient alternative. Roughly speaking, release consistency guarantees a coherent view of *all* shared data at synchronization points. This can have an adverse effect on performance especially in high-latency message-based environments.

Interactive client-server applications can often use consistency mechanisms that are more flexible, with client-controlled consistency and object sharing. For instance, some clients might require updates to shared objects at regular intervals instead of whenever the object is modified, while others might require updates whenever the object is modified "by a certain amount". In other words, such applications may tolerate stale data based on a temporal or change-based

⋆ This work was supported in part by NSF grants CDA–9401142, CCR–9702466, and CCR–9705594; and an external research grant from Digital Equipment Corporation.

criterion, thereby reducing communication overhead and improving efficiency. Since traditional shared memory environments do not provide such alternatives, developers of interactive applications are forced to use cumbersome and unsuitable programming interfaces in order to achieve efficiency.

InterAct is a runtime system that provides both efficiency and ease-of-use. It presents the user with a transparent and efficient data sharing mechanism across disparate processes. This feature enables clients to cache relevant shared data locally, enabling faster response times to interactive queries. Further, InterAct provides flexible client-controlled mechanisms to map and specify consistency requirements for shared objects. The consistency specifications are exploited by our system to enhance application performance. We evaluate our system on an interactive data mining application. Our results show upto a 3-fold improvement in client side response times to interactive queries.

The rest of this paper is organized as follows. Section 2 describes our overall goals, and the design of the runtime interface. Section 3 describes the implementation details of maintaining address independence and enforcing consistency requirements. Section 4 describes our experimental results. Section 5 describes related work. Our conclusions and on-going work are outlined in Section 6.

2 The Runtime Interface

In order to accomplish its goal of transparently supporting interactive applications, Interact must provide an interface that:

- Defines a data (structure) format for shared objects that is address space and architecture independent.
- Identifies, defines, and supports the types of consistency required.

We illustrate the design using the interactive data mining domain in particular an association mining system [12]. Data mining algorithms typically operate on large data sets and are compute intensive. The process is also largely repetitive, with the user perturbing the input parameters to a given task in order to arrive at the desired result. Due to the large datasets involved, some functions must be performed on the server since they would otherwise involve huge amounts of communication with the client. However, the interactive querying component is naturally performed on the client. This creates a natural split in work allocation. An efficient split entails sharing data summaries between client and server. This sharing can improve interaction efficiency while potentially reducing the server load.

In Figure 1, we describe a general-purpose interactive mining algorithm mapped onto InterAct. The client has mapped an array, a DAG, and a list onto the virtual shared dataspace. An element in the list points to the DAG object, represented by the dotted connection. Once mapped, a client can synchronize the local version of the data structure with the server at appropriate points. The server is responsible for creating and updating the data structures in the virtual dataspace. InterAct transparently handles all consistency and sharing-related updates as well as client-server communication.

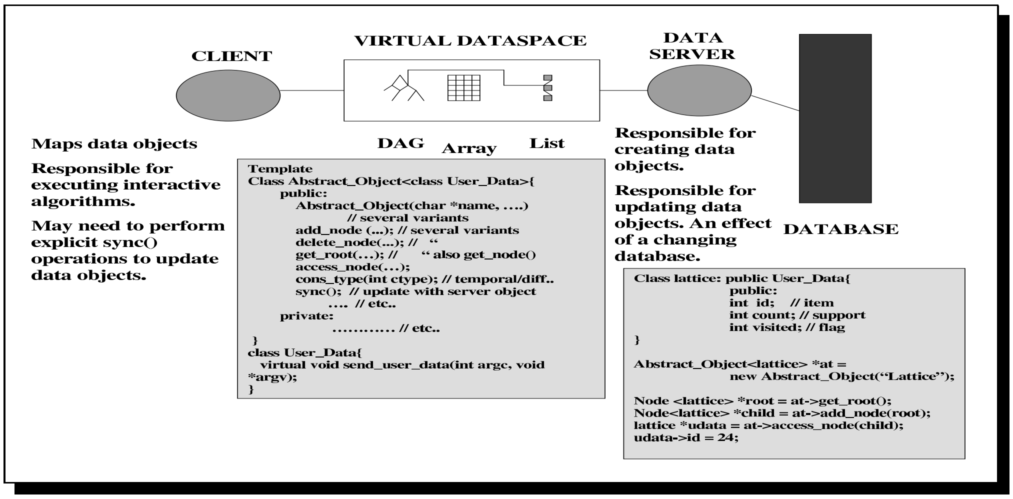

Fig. 1. Interactive Client-Server Mining

2.1 Data Declaration

In order to make the system both address-space and architecture-independent, the runtime system must define a suitable format for data representation. Our implementation relies on the use of C++ and some programmer annotation to identify the relevant information about shared data to the runtime system.

In Figure 1, we describe the current interface available to the user within the two greyed rectangular regions . The left-hand side is the InterAct object template/interface. Our interface is essentially a set of template classes with predefined functions for creating objects (`Abstract_Object`), defining (`User_Data`), adding (`add_node`), and accessing (`access_node`) nodes within an object, and functions for synchronizing (`sync`) and defining the required consistency type (`cons_type`). The right-hand side represents a sample declaration and usage of an InterAct object. For the association mining application each node contains 3 variables as shown in the the figure. The application creates an abstract object called “Lattice” (the DAG in Figure 1) consisting of such nodes. It then creates and accesses nodes in the lattice.

2.2 Consistency Types

As mentioned in the previous section our API includes mechanisms to express desired client consistency types, and an explicit way to make data consistent. The API also requires that changes to objects are made within explicitly identified *intervals*, which are similar to a transaction start and commit.

Sharing and caching is facilitated in our system by the provision of several consistency guarantees. The following consistency types are defined:

- **One Time Updates** specifies a one-time request for data by the client. No history need be maintained. This is the default consistency type.
- **Polled Updates** specifies that the client will make requests for updates when necessary.
- **Immediate Updates** specifies that the server should notify the client whenever there are any changes to the mapped data in any interval.
- **Diff Content Updates** specifies that the client must be updated if the data changes by a specified amount.
- **Temporal Updates** specifies that the client must be updated every x units of time. In other words, the data cannot be out of sync for $> x$ time units.

The different consistency types can be exploited by our system to enhance application performance. For instance, in some data mining applications, where a visual map is evaluated over equispaced time points, changes to the visual map need not be communicated immediately .Identifying this requirement using **Temporal Update** allows the runtime to make this optimization. In the process of discretization, it has been shown that small changes to the probability density function estimate does not severely affect the result quality. Identifying this property using **Diff-Content Update**, indicating that the application can live with stale data, allows the runtime to reduce the frequency of updates. Similarly, knowing that a client requires a **Polled Update** as opposed to a **One Time Update** tells the runtime system that shipping object changes might be more beneficial than resending the object on each poll.

3 Implementation Issues

The key to the efficient support of sharing is efficient address translation while maintaining address independence, efficiently identifying modifications to an object, and minimizing when these changes are communicated. We address each of these issues here.

3.1 Address Translation

InterAct objects request memory in page chunks using a segment-based memory placement library [9]. The library, which uses the Unix malloc routines, provides mechanisms to control memory placement policies for dynamic data structures. Using the library functions, we can identify the pages that belong to a given **object**. An **object** refers to a recursive data structure such as a tree or graph. Each such **object** is composed of a set of **nodes** that may be interconnected. Determining the address of a particular node within an object is similar to array indexing, since the size of each node is fixed. Communication is also simplified, since we communicate changes to the object at the granularity of a node.

Like any distributed object system, our interface identifies any pointers and their associated types to the runtime system in order to provide address-independence. During object creation, the object's internal representation is divided into data and connection pages. Data pages contain all the data information for an object. Similarly, connection pages store the actual connection information for nodes within an object. Information in the connection pages identify the object (in the case of inter-object pointers) and the node within the object in terms of an index, making address independence feasible. Separating connection and data information, co-locates all the pointers, enabling the runtime to perform rapid address translation.

3.2 Object Modification Detection

The technique we use to detect modifications is similar to that used by multiple-writer page-based software distributed shared memory systems [4, 2]. At the beginning of every interval, all relevant object pages are marked read-only using the *mprotect* VM call. When a processor incurs a write fault, it creates a write notice (*WN*) for the faulting page and appends the *WN* to a list of *WN*s associated with the current interval. It simultaneously saves a pristine copy of each page called a *twin* and enables write permissions [4]. The *twin* serves two purposes. First, if a client request comes in during an interval, data is delivered from the twin, thereby ensuring atomicity. Second, at the end of an interval the *twin* is used to identify the nodes modified within an interval.

When an interval completes, all objects that have been modified within the transaction interval (identified through the *WN* list) increment their associated object timestamp. Each object has an associated timestamp map. A timestamp map contains an entry for each node indicating the last time at which it was modified. The timestamp map for all the nodes contained in pages in the *WN* list are updated. Work is thus limited only to those pages that are actually modified. The update is done by comparing the page to its *twin* to determine the nodes that have been modified.

When asked for changes by the client, the object compares its timestamp map entries for each node against the last time the client has been appraised of object updates. The result of the timestamp comparison is a run-length encoding of the node data and node connections that have been modified, which constitute the *diff* of the object. Header information also specifies the object identifier, and number of node data and connection updates.

On the client side, we maintain information corresponding to the objects that are mapped and where they are stored in local memory. On receiving a diff message, we update the corresponding object by decoding the header to determine the object identifier. This object identifier is then used to determine the local location of the object. Data and connection information for nodes within the object are similarly address independent.

3.3 Consistency Maintenance

The runtime system optimizes data communication by using the consistency types specified by the user to determine when communication is required. The goal is to reduce messaging overhead and allow the overlap of computation and communication. Implementation of the **Immediate Update**, **Polled Update**, and **One-Time Update** consistency guarantees are fairly straightforward. **One-Time Update** does not require any meta-data maintenance. **Immediate Update** and **Polled Update** are implemented by having the client send the most recent timestamp of the object it has seen. Only those nodes with timestamps greater than this value are communicated. **Temporal Updates** are supported by having the runtime system on the client's side poll for updates at the intervals defined by the user. To keep track of **Diff Content Updates**, an additional element in the client timestamp array maintains a cumulative count of nodes modified since the last client update. If this cumulative count exceeds a preset (by the user) number, the client is sent an update. For this case the server has to maintain the last timestamp seen by the respective clients.

4 Experimental Evaluation

We evaluate our framework on the interactive association mining algorithm, used as a running example in this paper (Figure 1). In this application, the data server is responsible for creating a shared data structure (itemset lattice [12]) based on the local database. This lattice is subsequently mapped by the client. The client executes some non-trivial interactions on the lattice, as described in previous work [1]. The server updates the mapped data structure corresponding to changes in the database (which we simulate), and commits these changes. Updates are transmitted to the client on a client poll.

In our experimental setup, the server was running on a 143 MHz Sun UltraSparc machine, with 128 MB of memory. We evaluated the framework on 3 separate client configurations. **Client1** was a 50 MHz Sun SPARCstation-LX with 24 MB of memory, **Client2** was an 85 MHz Sun SPARCstation-4 with 32 MB of memory, and **Client3** was a 143 MHz Sun UltraSparc with 64 MB of memory. Our network was a 10 Mbps Ethernet.

Communication Performance: We evaluated the effectiveness of using diffs (with **Polled_Updates**) as opposed to resending the entire object (using **One_Time_Send**). We found sending diffs could be 10-30 times faster than resending the entire object for reasonable sized changes. The size of the diff message is much smaller, resulting in reduced communication overhead. The gains due to reduction in communication cost also reflect the reduction in overhead from client-server flow control due to finite buffer sizes.

Client Info	Client Side	Exec-Server	Speedup
Client1	4.74	6.0+0.54	1.4
Client2	1.65	3.8+0.54	2.6
Client3	0.83	1.7+0.54	2.7

Table 1. Interactive Performance (times in seconds, object sizes in bytes)

Interactive Performance: In order to estimate the benefits of permitting interactivity (through client side caching), in Table 1, we compare executing a mining query on the locally mapped data structure (Client-Side) as opposed to executing it on the server side and shipping the results to the client (Exec-Server[communication time + execution time]). For this particular query, the response time is reduced by about a factor of two when executing on the client-side. The faster the client, the bigger the speedup, since the difference in raw query execution times between client-side execution and server-side execution is reduced, resulting in higher relative communication overhead.

Runtime Overhead: We evaluated the overhead imposed by our system on the server side during normal execution without client connections. To evaluate this overhead we compared the test application written using the **InterAct** interface against a program written using standard C++. We found the runtime overhead imposed by adhering to our framework, for the above interactive experiments, to be less than 6%.

5 Related Work

Remote Procedure Call (RPC) mechanisms, and Remote Method Invocation (RMI) mechanisms have been used for building client-server applications. Distributed object systems such as Emerald [8], Rover [7], and Globe [11], also permit object sharing in the presence of replicated objects and object migration. Clearly, InterAct shares some features with such systems, such as support for address-independent objects (serialization/marshalling), and a need to identify where the pointers are and what they point to. However, in these systems, shared objects can be kept consistent only in one of two ways: modifications result in the entire data structure being communicated, or object logs keep track of method invocations and these methods are re-executed on object replicas. The former results in extra communication, while the latter may not be possible for interactive applications if the methods require data available only on the server side as in data mining applications. Our system provides a transparent mechanism to identify modifications made to these data structures, thereby reducing data communication.

Computer Supported Collaborative Work (CSCW) [5] systems share some of the features of our system (supporting interactive sessions across disparate systems, update notification etc.). However, most such applications support only a particular aspect of work, e.g., informal communication, distributed meetings, document co-authoring etc. This leads to a proliferation of isolated tools with little or no inter-operability.

Our work derives some of its characteristics from the distributed shared memory domain. In particular, the way we compute changes to our objects is a hybrid combination of the approaches described in TreadMarks [2] and Midway [13]. Orca [3] is similar in some respects to our work, particularly in that it provides an address independent object format similar to ours. Beehive [10] is a distributed shared memory system that also supports a temporal notion of consistency called delta consistency. This is similar to temporal updates in InterAct.

Our system differs from the above distributed shared memory approaches in that it supports sharing across disparate processes. Further, InterAct differs from these related approaches in its flexible supports for relaxed consistency types on a per client basis rather than on a per application basis. For instance, in InterAct, two clients can map the same data structure using a different consistency type.

6 Conclusions

We have described a general framework that supports efficient data structure sharing with client-controlled consistency for interactive client-server applications. The interface enables clients to cache relevant shared data locally, enabling faster response times to interactive queries. The interface eliminates the complexity of determining exactly what data to communicate among clients and servers, as well as when that data must be communicated. We are in the process of using the framework to develop applications in the interactive data-mining domain.

References

1. C. Aggarwal and P. Yu. Online generation of association rules. In *ICDE*, Feb 1998.
2. C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb 1996.
3. H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE TSE*, Jun 1992.
4. J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM SOSP*, Oct 1991.
5. Prasun Dewan. A Survey of Applications of CSCW Including Some in Educational Settings. *ED-MEDIA*, pages 147–152, Jun 1993.
6. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th ISCA*,May 1990.
7. A.D. Joseph, A.F. deLespinasse, J.A. Tauber, D.K. Gifford, and M.F. Kaashoek. Rover: A toolkit for mobile information access. In *15th SOSP*, Dec 1995.
8. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM TOCS*, 6(1):109–133, Feb 1988.
9. S. Parthasarathy, M. J. Zaki, and W. Li. Memory placement for parallel data mining on shared-memory systems. To appear in *KDD*, Aug 1998.
10. A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in beehive. In *PROC of the 9TH SPAA*, Jun 1997.
11. M. vanSteen, P. Homburg, and A.S. Tanenbaum. The architectural design of globe: A wide-area distributed system. In *Technical Report (Vrije University) IR-431*, Mar 1997.
12. M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, Dec 1997.
13. M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. Software write detection for distributed shared memory. In *PROC of first OSDI*, pages 87–100, Nov 1994.