Shared State for Client-Server Mining *

***S. Parthasarathy*[†], *and S. Dwarkadas*[‡]**

1 Introduction

For many organizations the explosive growth in data collection techniques and database technology has resulted in large and dynamically growing datasets. These organizations are increasingly turning to data mining, the process of extracting useful information from such datasets. These datasets are typically in a remote repository accessible via a local or inter-network. Despite advances in processing speed and networking technology remote data mining is difficult because of the conflicting requirements imposed by the size of the data involved and the interactive aspect of data mining.

The size of the datasets prohibit transferring the entire data to the remote client(s). In addition, data mining is often an iterative process with the user tweaking the supplied parameters according to domain-specific knowledge. This compounds the problem of increased response times due to network and server delays. We have shown [32] that often these applications can be structured so that subsequent requests can operate on relatively small summary data structures. Once the summary structure is computed and communicated to the client, interactions can take place on the client without further communication with the server.

The summary is based on the snapshot of the actual data at any point in time. If the data is dynamically being modified, the summary is likely to change. In this scenario, the client's copy of the summary structure must be kept up-to-date. Traditional realizations of this communication employ some form of message passing or remote procedure call (RPC) in order to keep data coherent, are rather cumbersome, and can be inefficient. Programming ease concerns suggest the need for an abstraction of shared state that is similar in spirit to distributed shared memory (DSM) semantics. However, even the most relaxed DSM coherence model (release consistency [17]) can result in a prohibitively large amount of communication for the type of environment in which data mining may typically be performed. These

*This work is supported in part by NSF grants EIA–9972881, CCR–9702466, CCR–9705594, and CCR-9988361; and an external research grant from Compaq.

†CIS Department, Ohio-State University. Email: srini@cis.ohio-state.edu

‡CS Department, University of Rochester. Email: sandhya@cs.rochester.edu

applications can often accept a significantly more relaxed—and hence less costly—coherence and consistency model, resulting in excellent performance gains. Overall system performance can be further improved by allowing each client to specify the data shared as well as the coherence model required for its needs.

In Section 2, we describe our overall system design goals and then discuss related work in Section 3. In Section 4, we describe our runtime framework, a system called **InterAct**, which allows efficient caching and sharing of data among independently deployed clients and servers. InterAct supports data sharing efficiently by communicating only the modified data, and by allowing individual clients to specify relaxed coherence requirements on a per data structure basis. We demonstrate the utility of the system using several applications from the interactive data mining domain, described in Section 5, These applications are structured so that the server is responsible for creating the data structure(s) (storing the summary), mapping them to a *virtual shared dataspace*, and subsequently keeping them up-to-date. The client can then map the data structure(s) from the *virtual shared dataspace* under an appropriate coherence model. Experimental results (Section 6) show that executing queries using the appropriate summary structure can improve performance significantly; up to a *23-fold improvement* in query execution times was observed. When the clients cache the summary structure using relaxed coherence models, we also observed *several orders of magnitude reduction in update costs.* Furthermore, for these applications, using such relaxed models does not significantly affect the quality of the results (we observed <2% degradation in result quality).

2 Design Goals

In order to accomplish the goal of efficiently providing shared state in a distributed environment, the runtime system must provide an interface that defines a mechanism to declare shared data that is address space independent and persistent (so that clients can join and leave at any time). In addition to the above minimal requirement for sharing, data mining applications have several properties that can be exploited, and key needs that ideally must be supported.

First, since clients have differing needs in terms of how up-to-date a copy of the data is acceptable, the system must identify, define, and support different relaxed coherence models that may be exploited for application performance. This feature of *client-controlled coherence* is similar to the notion of quasi-caching [4].

Second, many data mining applications require the capability of obtaining a *consistent* version of a shared data structure at any time, even in the presence of an on-going update to the data. We refer to this feature as *anytime updates.*

Third, many data mining applications traverse these summary structures in an ordered manner. Different clients may have different access patterns depending on the kind of queries processed. This feature requires that the system export programmer-controlled primitives that allows data to be remapped or placed in local memory in a manner that mirrors how the data is likely to be accessed by a given user or client. We refer to this feature as *client-controlled memory placement.*

Fourth, the shared data, although significantly reduced (summary), can still be quite large, so re-sending it on each update can cause significant delays on a busy network. It is therefore important for the runtime system to *identify which* *parts of the shared data have been modified* since the client's last update. Only the changes need be sent to the client on an update. In Section 4, we describe how our system, InterAct, can support the above data mining requirements while providing a general interface for a large class of applications.

3 Related Work

There is a rich body of literature studying the issues in caching and data sharing in many different computing environments. Distributed object-based systems [6, 24, 23, 14, 34, 39, 41, 7, 16], all support the basic requirement of sharing address-independent objects. However, update propagation in such systems (typically supported either by invalidate and resend on access or by RMI-style mechanisms) are inefficient (re-sending a large object or a log of operations (RMI)) and often infeasible[1] for data mining applications. Distributed shared memory systems [3, 5, 26, 35, 33, 20, 22, 43] all support transparent sharing of data amongst remote processes, with efficient update propagation, but most require tight coupling of processes with sharing that is not address-independent. None of the above systems support flexible client-controlled coherence, client-controlled memory placement (due to their address-dependent nature), or anytime updates.

Quasi-Caching [4] is very relevant to the topic of maintaining client-controlled coherence between a data source and cached copies of the data. Quasi-Caching assumes that each object has a computer that stores the most recent version and other computers store quasi-replicas that may diverge. They considered both time-based and scalar value based divergence (coherence) models. In their work, the allowed divergence is specified by the user in a fixed manner. The quasi-caching work describes when to update a client's cached copy but does not deal with the issue of how to do so efficiently. Furthermore, this work also does not support dynamically modifying the coherence model, or client-controlled memory placement.

Computer Supported Collaborative Work (CSCW) systems [37, 15, 11] share some of the features of our system (supporting interactive sessions across independent and potentially heterogeneous systems, update notification, etc.). However, most of these systems are tailored to a specific application, like cooperative engineering design, or distributed meetings. This has lead to a proliferation of isolated tools with little or no inter-operability.

There has also been some recent work on distributed data mining systems. The Kensington [19] architecture treats the entire distributed data as one logical entity and computes an overall model from this single logical entity. The architecture relies on standard protocols (such as Java Database Connectivity (JDBC)) to move the data. The Intelliminer [38] and Papyrus systems [18] are designed around data servers, compute servers, and clients as is the system presented in this work. All of the above systems rely on a message-passing-like interface for programming distributed data mining applications. InterAct provides a shared-object interface with features such as client-controlled coherence, memory placement and anytime updates. In this paper we limit ourselves to evaluating InterAct on client-server data mining applications but we should point out that InterAct can be used for a broader class of distributed applications as well.

[1]Especially if the methods require data available only on the server side

4 Runtime Framework

4.1 Interface

Shared data in **InterAct** are declared as *complex-objects*. Complex-objects are composed of *nodes* that may be linked together. Nodes may be C-style `structs`, basic types, or some predefined InterAct types. Complex-objects could include recursive data structures such as graphs, trees, lists, arrays or collections of nodes. In Figure 1, we describe a general-purpose interactive mining algorithm mapped onto *InterAct* . In this example, the client has mapped three complex-objects, an array, a directed acyclic graph (DAG), and a list representing the shared data summaries, onto the virtual shared dataspace. An element in the list points to the DAG, represented by the connection. The server is responsible for creating and updating these data summaries. The client specifies a coherence model when mapping the summaries, synchronizes when required, and is responsible for the interactive querying component.

In Figure 1, we describe the current interface available to the user within the two grayed rectangular regions. The left-hand side is the *InterAct* interface. Our interface is essentially a set of template classes with predefined functions for creating a complex-object (`new InterAct_Object`), remapping (`remap_object`, not in figure) it in memory in a locality-enhancing manner, adding (`add_node`), and accessing (`access_node` - so that the pointer manipulated is independent of any additional bookkeeping data present per node) nodes within a complex-object, and functions for synchronizing (`acquire/release read/write lock` [2]) and defining the required coherence type (`cons_type`). The User_Data class is used as a base to define what a node (`Node`) contains. The node may be composed of basic data types and pointers to other complex-objects. The interface requires the user to identify these special pointers (`object_ptr`, not in figure) to the system. The right-hand side represents a sample declaration and usage of an *InterAct* complex-object called **Lattice** (the DAG in Figure 1) consisting of user-defined nodes (each node contains 3 variables as shown in the figure). The sample code then creates and accesses nodes in the lattice. The process to first create (in this example the server) a complex object, becomes the default manager for that object and places the object in the shared space.

In InterAct, every complex-object moves through a series of consistent states, or versions. When a client first maps a shared complex-object, it specifies the desired coherence model. InterAct obtains a copy of the complex-object from the server on the first client access. At the beginning of each semantically meaningful sequence of operations, the client performs a synchronization operation, during which the system ensures that the local copy of the complex-object is "recent enough", as determined by the specified coherence protocol. If not, it obtains a new version from the server.

One of the principal innovations in the runtime system is the provision of the ability to allow each individual process to determine when a cached copy of a complex-object is "recent enough", through the specification of one of a set of

[2] Note that our implementation of a write lock is *relaxed*, in the sense that readers are permitted during a lock held in write mode.

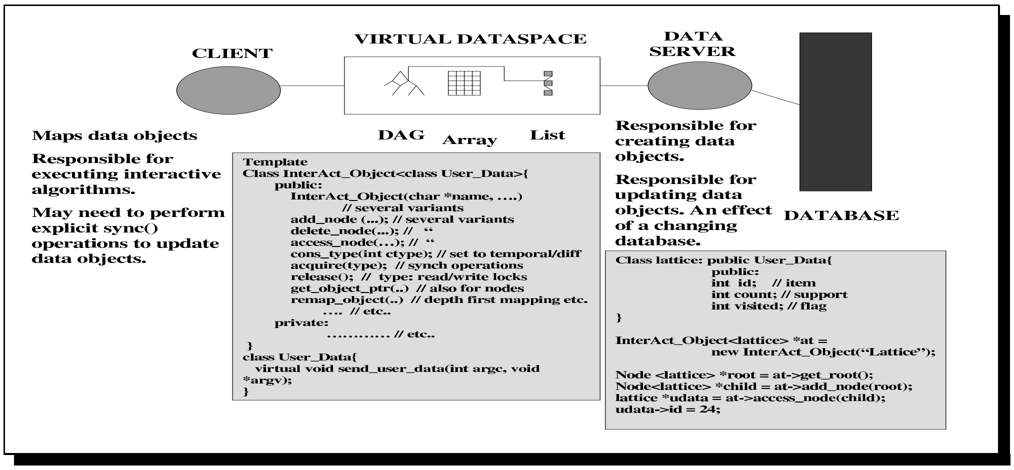

Figure 1. *Interactive Client-Server Mining*

highly relaxed coherence models. Changes to complex-objects must be made using mutually exclusive access. As long as applications adhere to these synchronization requirements, *InterAct* transparently handles all client-server communication including intra-object consistency and coherence maintenance.

The coherence models are motivated by the ability of a large number of data mining and other interactive applications to tolerate a certain level of data staleness. In all cases, the interface involves the use of synchronization in order to bring data up-to-date, or to make modifications to the data. The API allows processes to acquire a read or a write lock on shared data. A read lock guarantees that the shared data is up to date subject to the coherence model requested. A write lock always guarantees strict coherence.

In order to provide client-controlled memory placement, our API provides primitives by which each process (clients or servers) can locally remap nodes within a complex-object to improve spatial locality. *InterAct* transparently handles the remapping as a byproduct of supporting address independence.

4.2 Memory Management and Access

Like any distributed object-based system, our interface identifies any pointers and their associated types to the runtime system in order to provide address independence. During complex-object creation, the complex-object's internal representation is divided into data and connection pages[3] (this division is transparent to the user as long as the defined templates are used to declare and access shared data).

Data pages contain all the nodes for a complex-object. Nodes are created and allocated as fixed-size structures so that array index arithmetic [4] may be used to access them efficiently (allowing variable sized nodes would simply involve a slightly

[3]Each complex-object is mapped to a disjoint set of pages. which enables our system to transparently detect changes to objects using virtual memory hooks. Since, we are dealing with applications where the complex-objects are reasonably large relative to the size of a page, this does not result in memory wastage.

[4]Note that since complex-objects can dynamically change in size, all the pages for a complex-object need not be contiguous, so a slightly modified form of array indexing is needed.

less efficient access mechanism to the node). Separating connection information for a node (number of nodes linked to) allows the number of links to be variable and enables us to use array arithmetic on the data pages. Each node contains a single pointer into the connection pages that identifies the set of nodes that node links to. Information in the connection pages identify the node within the complex-object in terms of an index, making address independence feasible. Separating connection and data information also co-locates all the pointers, enabling the runtime system to perform efficient pointer swizzling [42].

Laying the data out in semi-contiguous order facilitates efficient address to node mappings as long as node sizes are fixed, resulting in fast identification of changes to a complex-object as outlined below in Section 4.3. The use of node identifiers coupled with the above scheme also enables fast node to address mapping that permits us to update node changes rapidly as well as maintain mappings independent of server mappings, as outlined below in Section 4.7.

4.3 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 [5, 8], except that we use a node as the granularity at which we detect modifications. At the start of every acquire of a write lock (see Figure 2), all relevant complex-object pages are marked read-only using the *mprotect* virtual memory system 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, or region encapsulated by an acquire and a release. It simultaneously saves a pristine copy of each page, called a *twin*, and enables write permissions [8]. When the lock is released, the *twin* is used to identify the nodes modified within the interval.

At the release, all objects that have been modified (identified through the *WN* list) increment their associated object timestamp (or *version* number). These objects are efficiently identified since our *WN* list is maintained as a hash table containing the <page address, object identifier> pairs. Since we ensure that complex-objects reside on separate pages, a write to a page corresponds to a write to a single complex-object. Modified nodes are identified by comparing the modified pages to their *twins*. Comparison is thus limited only to those pages that are actually modified. These modified nodes are then communicated to the object manager along with the latest version number in order to keep the manager's copy up-to-date. The object manager has a timestamp (or version) map associated with each object. A timestamp map contains an entry for each node indicating the last time it was modified (see Figure 2). Upon receiving modifications, the manager updates its copy of the data as well as of the timestamp map.

4.4 Updating an Object

When asked for changes by the caching process, the object manager (server) compares its timestamp map for the complex-object against the last time the client has been appraised of an update. 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 complex-object (Figure 2). Header information also specifies the complex-object identifier, and number of node data and connection updates.

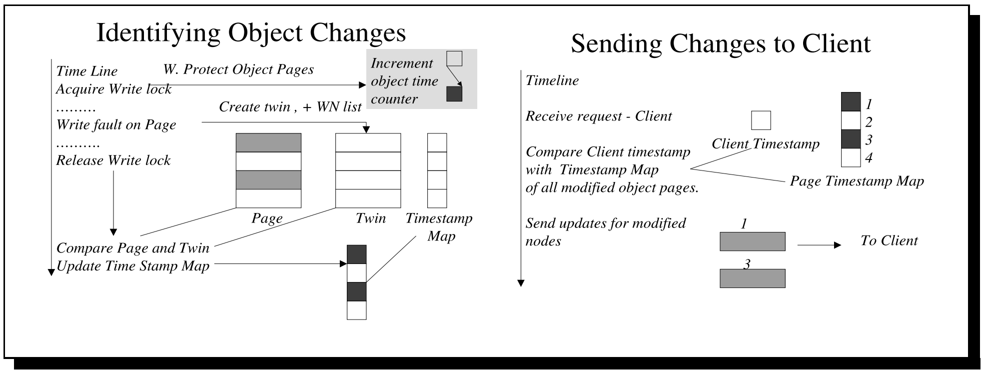

Figure 2. *Efficient Shared Data Updates*

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.

The issue of containment (or aggregation) in an object-oriented design is somewhat complicated, mostly because the definition of the term isn't universally agreed upon. It is an important issue as it is the primary means of supporting lazy deep copying efficiently. In our current implementation, when a client maps a particular object, we update all nodes that belong to the particular object immediately. However, if a node within this object points to another object or another node within another object, that object is not copied immediately. It is copied lazily upon client access or client request. The *InterAct* interface implicitly gives the programmer control over what needs to be copied immediately and what can be lazily copied. All the programmer has to do is to create separate complex-objects in this case.

4.5 Generating Anytime Updates

If a client request comes in while a write lock on the object is held by the server or managing process, one approach would be to wait[5] till the server transaction commits before sending an update to the client. This may not be acceptable for applications requiring a quick response, especially when the lock is held for a long time. Our approach is to twin the entire object on creation. When the write lock is released, the object twin is updated using the *diff* mechanism described above. If a client request comes in during a write lock, the system returns the update from the twin rather than the original object. During the application of the *diff* on the twin, the system returns the update from the original object. This ensures that the client rarely has to wait for the requested data in practice. Also, one could potentially switch between the two approaches according to application or system requirements.

[5]In order to guarantee an atomic update, the data cannot be sent in an as is condition, as partial changes to the object may have occurred concurrently.

4.6 Coherence Maintenance

InterAct defines the following coherence models. **One-Time Coherence** specifies a one-time request for data (complex-object) by the client. No history need be maintained. This is the default coherence type. **Polled Coherence** indicates that the client may request a current version when desired. The server may then attempt to reduce communication requirements by keeping track of the staleness of the data cached by the client. **Immediate Coherence** guarantees that the client will be notified whenever there are any changes to the mapped data. **Diff-based Coherence** guarantees that no more than $x\%$ of the nodes comprising a complex-object is out of date. **Delta Coherence** [35] guarantees that the complex-object is no more than x versions out of date. **Temporal Coherence** guarantees that the complex-object is no more than x real-time units out of date. In all cases, x can be specified by the client. Section 5 details the use of the different coherence models by various data mining techniques.

The runtime system optimizes data communication by using the coherence model 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 Coherence**, **Polled Coherence**, and **One-Time Coherence** guarantees are fairly straightforward. **One-Time Coherence** does not require any meta-data (timestamp) maintenance, nor does it require the server to generate object diffs. **Polled Coherence** is implemented by having the client send the most recent timestamp of the object it has seen. Under **Immediate Coherence** the server keeps track of this information. Under both Polled and Immediate coherence only those nodes with timestamps greater than this value are communicated. The difference between Polled and Immediate coherence is that in the latter the server notifies the client when a complex-object has been modified while the former has no such notification protocol. This allows the runtime system under Immediate coherence to check for notification messages before issuing an update request to the manager (server) eliminating some communication traffic. The upside to Polled Coherence is that the server does not need to maintain client-specific state on a per complex-object basis.

Temporal Coherence is supported by having the runtime system on the client's side poll for updates every x time units, as defined by the user. To keep track of **Diff-based Coherence**, the server maintains a cumulative count of nodes per complex-object that are modified since the last client update. If this cumulative count exceeds a preset user-defined value (referred to as the *diff* parameter), the client is sent a notification (similar to Immediate Coherence) and a subsequent update. **Delta Coherence** is kept track of in a manner similar to **Diff-based Coherence**. In these cases as well, the server has to maintain the last timestamp seen by the respective clients.

4.7 Memory Placement

Different clients may have different mining agendas, leading to different data structure access. *InterAct* permits the clients to place the mapped data structure in memory in a locality enhancing manner by using the remap()[6] function. For ex-

[6] Clients need to execute this only once. Subsequent updates from the server are automatically handled correctly by our system's address translation mechanisms.

ample, if the structure is a tree and most of the client interactions are going to induce a breadth-first evaluation of the tree, then the tree can be placed in memory in a breadth-first fashion to improve cache locality. *InterAct* currently supports breadth-first, depth-first, and user-defined placement [29]. User-defined placement allows the programmer to define a condition that splits the nodes in an object into separate sets of contiguous memory.

5 Applications

In past work, we [28, 30, 31, 27] have shown that it is possible to design useful summary structures for several mining applications so that subsequent queries can operate on these summary structures rather than the actual data. Within our framework of remote mining, these summary structures are generated and kept up-to-date by the data server, and subsequently mapped and operated on by the client. In this work we simulate the updates on the server side according to real data and application update properties, described below.

Association and Sequence Mining: Given a database of transactions where each transaction consists of a set of items, association discovery[2] finds all the item sets that frequently occur together, and also the rules among them. Sequence discovery essentially involves association discovery over temporal databases. It aims to discover sets of events that commonly occur over a period of time.

For association and sequence mining, the summary structure we use is the Itemset Lattice [1, 27] (or Sequence Lattice [30]), which contains pre-mined patterns and the corresponding support[7] information. Responses to user requests typically involve computing a constraint-based subset of the entries in the lattice. Updates to the lattice are handled as described in [40]([30]). Each incremental update, reflecting new data, typically combines multiple actual transactions, for performance reasons [40, 27]. There are two possible kinds of updates to the summary structure based on the type of mining being performed. When mining is performed on the entire database, new transactions are usually only added to the database — we call these additive updates. When mining is performed on a window of transactions in the database, changes in the database result in almost as many additions as deletions to the window — we call these windowed updates. Both types of updates typically result in changes to anywhere from 0.1% to 10% of the summary data structure. This is because in typical scenarios the number of transactions being added or deleted is a small percentage (0.1%-1%) of the total number of transactions being represented (and in the case of sequence mining not all customers are part of each update), so the net impact on the summary is relatively low. Additive updates mostly result in modifications to support counts and a few pattern additions and deletions. However, for windowed updates, the changes tend to result in more associations being added and/or deleted. For these applications, since the user is usually interested in keeping track of less frequently occurring associations or sequences, a stricter coherence model such as **Polled Coherence** or **Immediate Coherence** is generally preferred.

Discretization: Discretization has typically been thought of as the partitioning of the space induced by one (say X) or more continuous attributes (base) into regions

[7]Support is the number of times the pattern occurs in the dataset.

(X < 5, X >= 5), which highlights the behavior of a related discrete attribute (goal). Discretization has been used for classification in the decision tree context and also for summarization in situations where one needs to transform a continuous attribute space into a discrete one with minimum "loss". Our application is an instance of 2-dimensional discretization (two base attributes, described in [28]).

Interactions supported include generating an optimal discretization (based on entropy or classification error), and modifying the location and number of control points (which partition the two-dimensional base attribute space). The summary structure required to support such interactions efficiently is the joint probability density function (pdf) of the base and goal attributes. This pdf is estimated at discrete locations. While several techniques exist to estimate the density of an unknown pdf, the most popular ones are histogram, moving window, and kernel estimates [10]. We use the histogram estimate described in [10]. The advantage of this estimate is that it can be incrementally maintained in a trivial manner (a histogram estimate is essentially the frequency distribution normalized to one). Moreover, the more complicated kernel estimates can easily be derived from this basic estimate [10]. Each update corresponds to one transaction and every update modifies exactly one entry in the array. Each update is simply a small perturbation on the pdf estimate and as such does not affect the quality of discretization significantly. Thus, this technique would benefit from using **Diff-based Coherence** without affecting the quality of the results. The *diff* parameter specifies the amount of data that needs to change before it is significant to the application.

Similarity Discovery in Datasets: This application computes and maintains the similarity between two or more datasets. Such measures of similarity are useful for clustering homogeneous datasets. In [31], we define the similarity between two datasets to be a function of the difference between the set of associations induced by them, weighted by the supports of each association. To compute and maintain the similarity between n datasets, the client maps the itemset lattices (containing the association patterns and their supports) from each of the distinct data sources and then computes the pairwise similarity measures.

It has been noted by others [9] that incorporating domain bias, via suitable interactions, in the similarity measure can be very useful. In this application, we support the following operations: similarity matrix re-generation after a data structure update, identifying influential attributes, and constraining the similarity probe set via constraint queries on the itemset lattices. Incrementally maintaining the association lattices has already been discussed above. However, since this application is more interested in general patterns, even if a large percentage of the mapped summary structure is modified over a period of several updates, it has been shown that the percentage change in the measured similarity is not significantly affected. The measured similarity directly correlates more to the magnitude of the change in data. This magnitude is not directly measurable without a large amount of overhead. However, the use of **Delta Coherence** captures this application's requirements by allowing the data to be several versions out of date without affecting result quality.

Application	Object Size	UPS	%Change	Change Type
Association Mining	3.3MB	0.5	4%	ADD/DEL
Sequence Mining	1.0MB	1	10%	MOD/ADD
Discretization	0.5MB	100	0.002%	MOD
Similarity(1)	0.5MB	5	10%	MOD/ADD
Similarity(3)	3X0.5MB	5	0.33%	MOD/ADD

Table 1. *Server Update Properties*

Properties of Server Updates We outline the exact nature of the updates[8] used for our experiments for each application in Table 1. The second column refers to the size of the summary structure when the server starts up. UPS corresponds to updates per second on the server side. The column labeled %Change corresponds to the percentage of nodes in the summary structure that are changed over a single update, and are representative of realistic workloads for each of the applications. For Similarity Discovery, we computed the similarity among four databases. Updates on one of these databases (Similarity(1)) had different properties from the other three (Similarity(3)). The last column in the table refers to the dominant change type of the given update. ADD refers to the fact that the update adds new nodes, DEL refers to the fact that the update deletes nodes, MOD refers to the fact that the update modifies existing nodes. The order in which change types appear in column five of the table are in decreasing order of dominance. For example, ADD/DEL refers to the fact that on average, executing the corresponding update results in more additions than deletions to the summary structure.

6 Experimental Evaluation

We evaluate our framework in a distributed environment consisting of SUN workstations connected by 10 or 100 Mbps switched Ethernet. Unless otherwise stated, the clients use a 100 Mbps link, and are 270 MHz UltraSparc IIi machines. Our server was a 333 MHz UltraSparc IIi machine with 8 processors.

6.1 Client-Side Caching

In this experiment, we evaluate the impact of caching the summary data structures on the client-side so that interactions may be performed locally. We compared the above setting with a typical client-server setting, where the client makes a request to the server, the server computes the result[9], and then sends the result back to the client (SSRC). In order to better understand the impact of server load, we also varied the number of clients serviced by the server from one (SSRC) to eight (LSSRC). Other factors that affect performance include the size of the shared data, and the speed of the client (which we varied). Results are presented in Table 2 for these scenarios under different network configurations. We varied the network configuration by choosing clients that are connected to the server via a 10 Mbps or a 100 Mbps Ethernet network.

[8] The datasets used are described in [32].

[9] Note that the results are likely to be smaller than the actual size of data shared. For each of the applications considered in this experiment, Associations, Sequences, Discretization, and Similarity, the average size of the results shipped by the server was 1.5MB, 0.25MB, 0.4MB and 0.75MB respectively.

Apps	Client-143Mhz			Client-270Mhz		
	CSC	SSRC	LSSRC	CSC	SSRC	LSSRC
Ass.	2.4	1.6(**4.05**)	2.5(**7.2**)	1.5	1.4(**2.5**)	2.3(**5.1**)
Seq.	0.58	0.55(**0.85**)	0.86(**1.35**)	0.35	0.5(**0.63**)	0.73(**1.18**)
Disc.	0.87	0.67(**1.35**)	1.08(**2.75**)	0.55	0.6(**0.94**)	0.98(**1.6**)
Sim.	0.35	0.55(**1.5**)	0.98(**2.7**)	0.11	0.37(**0.9**)	0.94(**2.4**)

Table 2. *Time (in seconds) to Execute Query: 100Mbps (10Mbps)Interconnect*

The results in Table 2 show that client-side caching is beneficial for all but a few of the cases. In particular, the following trends are observed. Client-side caching is more beneficial when: the network bandwidth is low (speedups from client-side caching under the 10Mbps configuration are larger (1.5 to 23) than the 100Mbps numbers (0.6 to 9)), the server is loaded (comparing LSSRC with SSRC), the client is a fast machine (270Mhz vs. 143Mhz), or the time to execute the query is low (similarity discovery vs. association mining). As expected, these results demonstrate that the lower the computation/communication ratio, the greater the gain from client-side caching.

6.2 Coherence Model Evaluation

In this section, we evaluate the benefits of using relaxed coherence models. Figure 3 reports the average synchronization time[10] for each of the applications under different coherence models. We measured the synchronization time over a window of several (35[11]) synchronizations and averaged this time. This average synchronization time in seconds is represented on the Y axis. The X axis corresponds to the number of clients in the experiment. In these experiments, all clients use the same coherence model.

As we can see from the top two graphs in Figure 3 (for association and sequence mining), our basic coherence model (*polled coherence*), significantly outperforms (10-30 times faster) resending the entire complex-object (*object resend*, achieved using one-time coherence), the strawman approach taken by most existing commercial object-oriented systems. The gains due to reduction in communication cost reflect not only the reduction in data communicated but also the reduction in overhead from client-server flow control due to finite buffer sizes. Immediate coherence performs several factors better than polled coherence for both of these applications. On breaking down the average synchronization times, we found that there are two reasons for this. First, under polled coherence, every synchronization request involves communicating with the server, even if there are no updates. Under immediate coherence, communication takes place only when there is an update on the server side. As a result, the number of messages sent under polled coherence is larger than the number of messages sent under immediate coherence. Second, in the case of sequence mining, the total amount of data sent under immediate coherence was also less than the total amount of data sent under polled coherence. The synchronization rate in this application is fairly close to the rate at which the

[10]The time for acquiring a read lock, to bring the complex-object up-to-date according to the desired coherence model.
[11]Going to larger window sizes did not affect the average synchronization times much for our workloads.

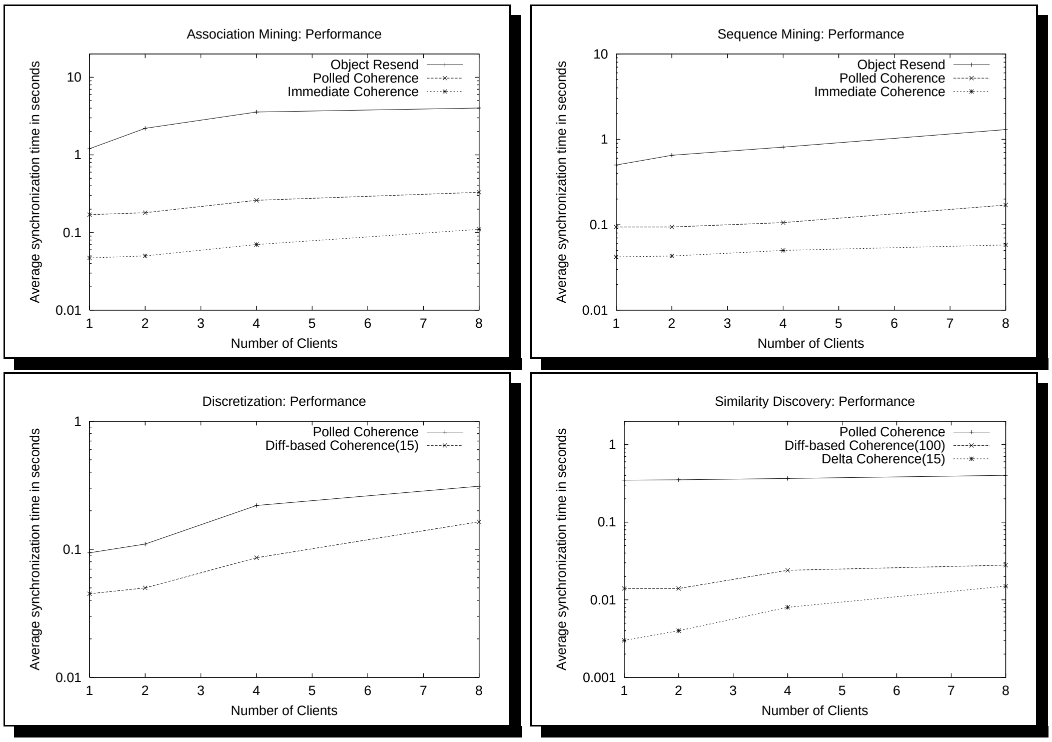

Figure 3. *Coherence Model Evaluation*

server modifies data. Hence, due to timing variations in receiving notifications with immediate coherence, some of the synchronization operations remain local and do not request updates. Also, this application primarily modifies existing nodes (see Table 1) and there is a significant overlap in nodes modified by back-to-back server updates. As a result, combining two server updates in one update message results in less data being communicated than when two separate updates are used.

While association mining and sequence mining benefit from sending updates rather than the entire complex-object, they both require the client to have the latest copy of the shared data. However, discretization and similarity discovery can tolerate some staleness in the interaction structure without losing much accuracy. Discretization can make use of the diff-based coherence model since we know that the quality of discretization is not affected by small changes in the shared data. Similarity discovery does not benefit as much from diff-based coherence since every server update is likely to modify roughly 10% of one of the summary structures, or 1600 nodes (see Table 1), which far exceeds the *diff* parameter that we use. The number in brackets for diff-based coherence in Figures 3 and 4 is the *diff* parameter, or the number of nodes that must be modified to trigger an update to the client (set to 15).

Similarity discovery benefits more from the delta coherence model due to the fact that the result quality is affected more by the magnitude of the change in data than by the amount of data that has changed. The magnitude of change per server update is small, while the number of nodes changed is about 10%. The

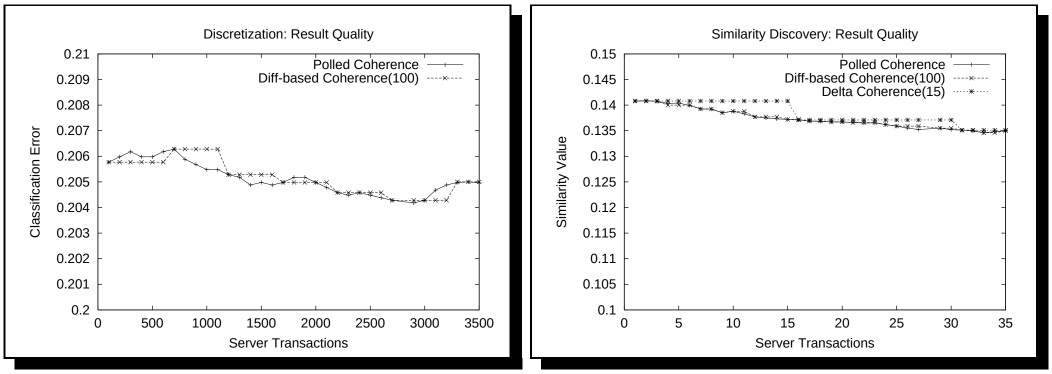

Figure 4. *Result Quality*

number in brackets for delta coherence is the maximum number of server updates (or versions, corresponding to releasing a write lock) between updates to the client. For these applications, we compare the average synchronization times against that with polled coherence. Referring to Figure 3, for discretization, we find that diff-based coherence, when using a diff parameter of 15, was on average twice as efficient as polled coherence. This difference increases as the diff parameter is increased. For a diff parameter of 100 we found that the average synchronization time is 15-20 fold better. Delta coherence is particularly effective for similarity discovery, outperforming polled coherence by three orders of magnitude at low server load.

In both of these applications there is a reduction not only in the number of requests but also in the total data communicated even when compared to polled coherence. The reduction is due to the applications' tolerance for staleness as specified by the relaxed coherence models.

Interaction Quality: The above results indicate that encoding knowledge about application behavior by choosing the appropriate coherence model can improve performance. However, we must also evaluate the corresponding loss in result quality for the application when using more relaxed coherence models. For each of the latter two applications, we plotted the result quality under the different coherence protocols over a period of time (demarcated by server updates). Figure 4 presents these results. For discretization, the result quality is represented by the classification error — the fraction of points in the space that are mis-classified. The plot for polled coherence represents the best achievable classification error for the algorithm. The plot for diff-based coherence (with a diff parameter of 100 as opposed to the 15 that we used in Figure 3) is the error obtained when using a relaxed coherence model. Clearly, for this application the loss in quality is not significant. In fact, it is off by less than 1% of the exact error at all instances in time even for such a high diff parameter. For similarity discovery, the result quality is represented by the similarity value, which represents the distance between two datasets. The similarity value using delta coherence is no worse than 2% off the value obtained using polled coherence at all instances in time.

6.3 Client-Controlled Coherence

An important contribution of our system is the fact that different clients may map the same shared structure using different coherence models. As an example, a client interested in similarity discovery involving a particular database could map the association lattice (as described in Section 5) of the database using the delta coherence model. The same lattice may be mapped by another client for the purpose of association mining using the polled coherence model. In this experiment, we considered the following configurations. In the polled configuration, all eight of our clients used polled coherence. In the delta configuration, all eight used delta coherence. In the mixed configuration, four used delta coherence and four used polled coherence.

We found the average synchronization time of clients under polled coherence in the *Mixed* configuration to be slightly lower due to reduced traffic, and that for clients using delta coherence to be slightly higher due to the extra traffic from the clients using polled coherence. The average synchronization times of clients under delta coherence in the *Mixed* experiment were two orders of magnitude lower than those for when all the clients mapped the data using polled coherence. If client-controlled coherence were not used, the server would have to adhere to the strictest coherence model for correctness, in this case, polled coherence, resulting in much reduced performance. Thus, by using the coherence model required by each client, the server is able to improve overall performance.

6.4 Effect of Modification Rate

In order to evaluate the impact of the server modification rate on our choice of coherence model, we modified the similarity discovery experiment described in the previous sections in the following ways. Two of the four lattices that we map have server update characteristics as described in Table 1, Similarity (1). For the other two lattices, we used the server update characteristics described under Similarity(3), where each server update modifies less that 0.1% of the data structure. Each lattice is maintained by a separate server process running on our 8-processor server.

We then evaluated the average synchronization time for one client in the system while varying the server updates per second. We varied *diff*, the diff parameter or the number of nodes that differ before an update is sent to the client, from 10 to 100 for diff-based coherence (see Figure 5). We also varied *delta*, the number of virtual time intervals between successive updates to the client, from 10 to 100 for delta coherence. The larger the diff/delta values, the lower the average synchronization time. The delta-based approach still does better for this application at a low server update rate since it minimizes the communication with the client. However, at larger update rates (crossover point 230 ups), diff-based coherence (diff_num = 100) begins to perform better than delta coherence. The reason for this is that at higher update rates, updates to the client are sent too frequently (in other words, a delta_num of 100 is too low) for all the complex-objects. However, since only two of the four lattices have a large percentage modified per update, for a *diff* value of 100, only these two lattices will cause updates to be sent to the client. The other two lattices are not modified at the same rate, resulting in lower average synchronization times when using diff-based coherence with a *diff* value of 100.

This experiment highlights the influence of server update properties on the choice of a coherence model, as well as the importance of being able to dynamically change coherence requirements as the application behavior changes. In order to be

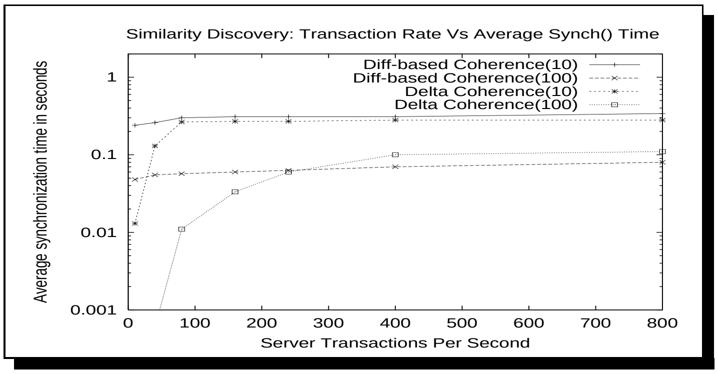

Figure 5. *Effect of Transaction Rate on Similarity Performance*

effective, choosing between relaxed coherence models is not only a function of what the application can tolerate in terms of data staleness, but also a function of how much and how often changes are made to the shared summary structure.

6.5 Locality Enhancements

Locality enhancing memory placement is especially useful when the client uses a predefined traversal of the shared summary structure. We illustrate its benefits using association mining, where we found that for different queries, a different mapping of the data structure presented the best results. For example, quantified associations uses a breadth-first traversal, including associations uses user-defined and depth-first traversal, and support subset associations uses user-defined traversal[12]. For these queries, we found up to a 20% improvement in execution times by remapping the data structure according to the best mapping strategy. We also evaluated the overhead imposed by the InterAct framework and found that the overhead (which is less than 5% for our application suite) is more than compensated by the improved memory locality.

7 Conclusions

We have described a general framework that supports efficient data structure sharing with client-controlled coherence for interactive and distributed client-server applications. The system supports a wide range of application domains, including visualization of scientific simulations and the remote tracking of images. In this paper, we have evaluated its performance on a suite of interactive data mining applications. The runtime interface enables clients to cache relevant shared data locally, resulting in faster (up to an order of magnitude) response times to interactive queries. The complexity of determining exactly what data to communicate among clients and servers, as well as when that data must be communicated, is encapsulated in the runtime system. Encoding knowledge about application behavior through client-controlled coherence enables further reduction in communication time by several orders of magnitude, since only those clients that require stricter coherence need use it. In addition, the runtime system supports anytime updates and client-controlled memory placement, both of which can help improve the performance of several data mining applications.

[12]These different association queries are commonly used in online association mining [1]

Bibliography

[1] C. Aggarwal and P. Yu. Online generation of association rules. In *IEEE International Conference on Data Engineering*, February 1998.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*. MIT Press, 1996.

[3] S. Ahuja, N. Carreiro, D. Gelernter, and V. Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[4] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM TODS*, 15(3):359-384, Sept. 1990.

[5] 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, February 1996.

[6] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, June 1992.

[7] M. Carey et al. Shoring Up Persistent Applications *Proc. of the 1994 ACM SIGMOD Conference*, 1994.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[9] G. Das, H. Mannila, and P. Ronkainen. Similarity of attributes by external probes. In *Proceedings of the 4th Symposium on Knowledge Discovery and Data-Mining*, 1998.

[10] L. Devroye. A course in density estimation. In *Birkhauser:Boston MA*, 1987.

[11] P. Dewan and J. Riedl. Towards computer-supported concurrent software engineering. In *IEEE Computer, Volume 26, Number 1*, 1993.

[12] A. Dingle and T. Partl. Web cache coherence. In *Proceedings of 5th WWW Conference (journal version: IJCN)*, 1997.

[13] D. Grunwald, B. Zorn and R. Henderson. Improving the cache locality of memory allocation. In *Programming Languages Design and Implementation*, June 1993.

[14] B. Liskov et al. Safe and efficient sharing of persistent objects in thor. In *SIGMOD*, 1996.

[15] G. Fitzpatrick, S. Kaplan, and W. Tollone. Work, locales and distributed social worlds. In *European Conference on Computer Supported Collaborative Work*, 1995.

[16] M. J. Franklin. Client Data Caching: A Foundation for High Performance Object Database Systems. Kluwer Academic Publishers, 1996

[17] 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 Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[18] R. Grossman, S. Bailey, S. Kasif, D. Mon, A. Ramu, and B. Malhi. Design of papyrus: A system for high performance, distributed data mining over clusters, meta-clusters and super-clusters. In *Proceedings of Workshop on Distributed Data Mining, alongwith KDD98*, Aug 1998.

[19] Y. Guo, S. Rueger, J. Sutiwaraphun, and J. Forbes-Millot. Meta-learning for paralle data mining. In *Proceedings of the Seventh Parallel Computing Workshop*, 1997.

[20] L. Iftode, C. Dubnicki, E.W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *High Performance Computer Architecture*, pages 14–25, February 1996.

[21] Y. Huang, R. Sloan, and O. Wolfson. Divergence Caching in Client-Server Architectures. IEEE Conf. on Parallel and Distributed Information Systems, 1994.

[22] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 213–228, December 1995.

[23] 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.

[24] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[25] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *3rd Intl. Conf. Information and Knowledge Management*, pages 401–407, November 1994.

[26] L. Kontothanasis, G. Hunt, R.Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *PROC of the 24TH ISCA*, June 1997.

[27] S. Parthasarathy. Active Data Mining in a Distributed Setting. PhD Dissertation, University of Rochester, 1999.

[28] S. Parthasarathy, R. Subramonian, and R. Venkata. Generalized discretization for summarization and classification. In *PADD*, January 1998.

[29] S. Parthasarathy, M. Zaki, and W. Li. Memory placement techniques for parallel association mining. In *Proceedings of the 4th Symposium on Knowledge Discovery and Data-Mining*, 1998.

[30] S.Parthasarathy,M.Zaki,M.Ogihara,and S.Dwarkadas. Incremental and interactive sequence mining. ACM Conference on Information and Knowledge Management 1999.

[31] S. Parthasarathy, and M. Ogihara. Clustering Homogeneous Distributed Datasets.. Fourth Practical applications of Knowledge Discovery and Data Mining (PKDD), 2000.

[32] S.Parthasarathy, S.Dwarkadas, and M. Ogihara. Active mining in a distributed setting. In *Workshop on Parallel and Distributed KDD systems*, 1999.

[33] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[34] M. Shapiro, S. Kloosterman, and F. Riccardi. PerDiS — A Persistent Distributed Store for Cooperative Applications. In *Proceedings of the Third Cabernet Plenary Workshop*, Rennes, France, April 1997.

[35] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in beehive. In *PROC of the 9TH SPAA*, June 1997.

[36] R. Srinivasan, C. Liang, and Krithi Ramamritham Maintaining Temporal Coherency of Virtual Data Warehouses. In IEEE Real-Time Systems Symposium (RTSS98), Dec. 1998.

[37] D. Sriram, R. Logcher, N. Groleau, and J. Chernoff. Dice: An object oriented programming environment for cooperative engineering design. In *AI in Enginnering Design, Vol. 3, Academic Press*, 1992.

[38] R. Subramonian and S. Parthasarathy. A framework for distributed data mining. In *Proceedings of Workshop on Distributed Data Mining, alongwith KDD98*, Aug 1998.

[39] D.B. Terry, M.M. Theimer, K. Peterson, A.J. Demers, M.J Spreitzer, and C.H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *15th SOSP*, Dec 1995.

[40] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. Incremental updation of association rules. In *KDD97*, Aug 1997.

[41] 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*, March 1997.

[42] P. R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *International Workshop on Object Orientation in Operating Systems*, Sept. 1992.

[43] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. Software write detection for distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 87–100, November 1994.