DataExchange: High Performance Communications in Distributed Laboratories

Greg Eisenhauer, Beth Schroeder, Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

Current communications tools and libraries for high performance computing are designed for platforms and applications that exhibit relatively stable computational and communication characteristics. In contrast, the demands of (1) mixed environments in which high performance applications interact with multiple end users, visualizations, storage engines, and I/O engines – termed 'distributed laboratories' in our research – and (2) high performance collaborative computing applications in general, exhibit additional complexities in terms of dynamic behaviors. This paper explores the communication requirements of distributed laboratories, and it describes the DataExchange communication infrastructure supporting their high performance interactive and collaborative applications.

1 Communication Substrates for High Performance Applications

Communication tools and libraries for high performance distributed computing have evolved substantially during the last few years, from systems enabling distributed parallel computing such as PVM[10], to industrial standards being actively implemented and improved such as MPI[9], to recent proposals for infrastructures that can facilitate the construction of nationwide, networked supercomputers. Our work contributes to such research by addressing several specific issues that arise whenever diverse networked machines are used in research and development settings or even in production environments, by single or multiple end users. The target applications we consider are those in which end users interact with powerful computational tools and with each other across heterogeneous networked machines, termed 'distributed laboratories'. The communication demands of such dynamic computational environments share some commonalities with traditional distributed systems as well as with the highly dynamic environment of today's web browsers and CORBA-based distributed object systems, but unfortunately these similar environments do not adequately

address the high performance needs of the some applications. Moreover, these environments' dynamic communication and computational behaviors are not addressed by standards for distributed high performance computing like MPI.

The needs of dynamic environments like distributed laboratories are the focus of recent research efforts on a variety of fronts. For instance, the LabSpace project at Argonne and at Northeastern University[12] is trying to create a virtual space in which scientists who are distributed across the nation can meet, talk, plan their research, and run their experiments. Support for the dynamic addition of clients is being included in MPI-2[8]. Similarly, the San Diego project KeLP[1] seeks to improve the manner in which accesses to remote data may be made more efficient by investigation of object-based data bases for data storage that enable computations to migrate toward useful data as well as by investigation of alternative, compiler-based approaches to linking interactive applications[18].

Our work complements the efforts described above by assuming that end users should be able to exploit the potentially high performance of vendor-provided implementations of standards like MPI when appropriate for their applications. In addition, when application requirements can be stated statically, users should be able to exploit available tools, such as specialized communication libraries (*e.g.*, fast messages[14]) or the automatic support for parallelization and inter-task communications offered HPF compilers . Given these assumptions, the 'Distributed Laboratories Project'[15] at the Georgia Institute of Technology is developing communication support that addresses the interactive and dynamic nature of high performance applications to enable their use in scientific investigations or in production environments, by single and by multiple end users. Specifically, the *DataExchange* communication library utilized in the implementation of several distributed high performance applications offers the following functionality:

- a publish-subscribe communications model,
- fast heterogeneous data transfer, and
- support for application-level diversity.

In the remainder of this paper, we characterize the communication characteristics of high performance applications when they are used in distributed laboratory settings. Next, we identify the requirements these characteristics impose on the underlying communication infrastructure. Using examples from the high performance interactive applications being developed at Georgia Tech, we also describe the DataExchange communication library. Experimentation with DataExchange across networked machines and with the sample application serves to evaluate the manner in which the identified communication requirements are met.

2 The Distributed Laboratory Environment

2.1 Characterizing Distributed Laboratories

In order to illustrate the challenges posed by distributed laboratory environments, we first consider a specific application. In particular, consider a large scientific simulation running

on a set of computational resources, such as a global climate model being developed for networked parallel machines[11]. This global model might interact with more detailed local climate, atmospheric or pollution models in order to enhance the accuracy of both the local and global models. These models may run concurrently on a variety of parallel and distributed computing resources, or, ideally, any one of the global or local models should be able to be replaced at runtime with a historical database containing information from previous model runs. In addition, model outputs may be processed using a variety of instruments, including specialized visualization interfaces or computational instruments performing calculations which derive from and expand upon the basic model results. Similarly, model inputs may be provided via other instruments that either utilize live satellite feeds or access stored satellite data on remote machines.

Consider the introduction of observers into this application scenario. Given the scale of such an application and the speed and complexity of its execution platform, it is easy to imagine that the simulation's progress and results are monitored by multiple scientists in distributed locations. The scientists' interests may vary from wishing to see the "big picture," to investigating in detail subsets of the simulation's output, to collaborating with each other via the computational instruments these models implement. The distributed application has additional components that assemble the information needed to drive various interactive displays, by gathering data from the distributed simulation and performing the analyses and reductions required for these displays. Some of these components may themselves have substantial computational or storage needs and require dedicated, additional resources of their own. They may also require access to additional information, as is the case for the atmospheric model's display with which end users compare observational (satellite) data with model outputs in order to assess model validity or fidelity. Moreover, since end users control the set of computational instruments, input and output components, and the displays, the current set of interacting computational instruments will change dynamically, driven by end users' needs or by the current needs of the running simulation. Figure 1 depicts a sample application configuration.

The previous paragraph demonstrates the dynamic nature of computational instruments and their connections. Another attribute of instrument interactions is the bi-directional nature of not only instrument-to-instrument communications but also the communications between instruments and end users and between multiple collaborating users. While it is clear that end users frequently interact with displays and with the analysis components used prior to display, it has become more common for users to also interact with the actual simulations while they are running, perhaps to *steer* them toward more appropriate data domains, to enable or disable certain ancillary computations, or to experiment with alternative settings for simulation parameters. The benefits of such dynamic interactions of end users with large-scale simulations are well known for discrete event simulations and have recently been found to offer substantial benefits to scientific and engineering users as well. In addition, it is well known that significant performance improvements may be derived for high performance applications from the runtime configuration of selected application attributes[5], from the online adjustment of their execution environment (*e.g.*, process migration), and from the adaptation of underlying operating system or communication facilities[2].

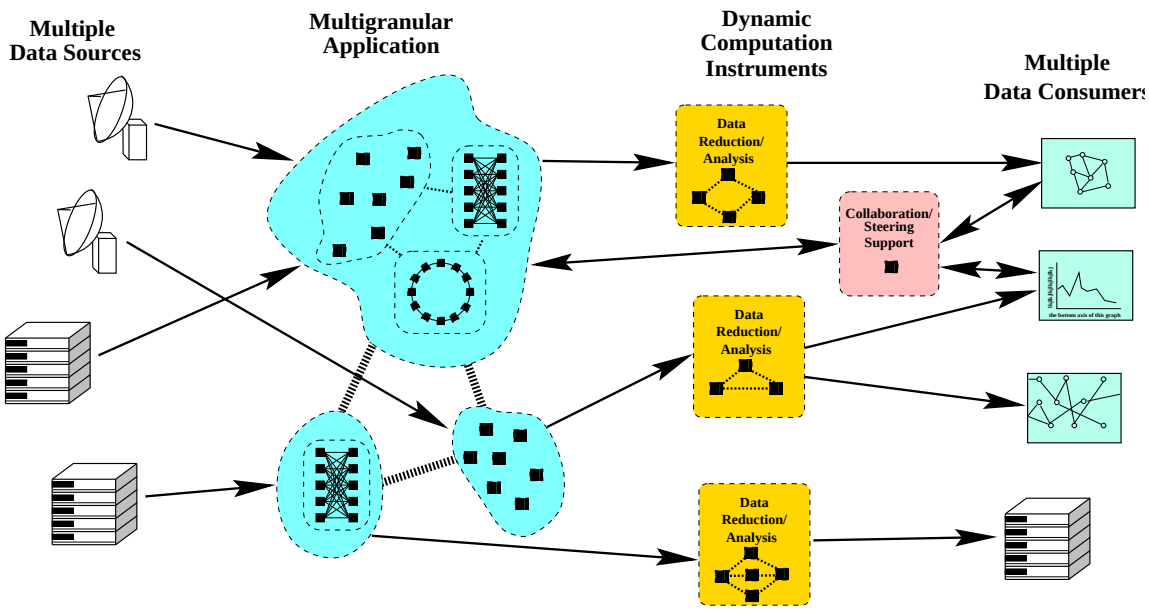

Fig. 1. A sample distributed laboratories application configuration.

Taken as a whole, this application scenario has requirements for data management and exchange that go well beyond those of a traditional parallel or distributed application. In particular, it presents a need for efficient high-performance data exchange coupled with data location and analysis in a highly dynamic heterogeneous environment. These communication needs are discussed in more detail in the next section.

2.2 Communication Requirements of Distributed Laboratory Applications

2.2.1 Flexible High-bandwidth Data Transfer

The need for high bandwidth and low latency in communications is common to most high performance applications, and those running in the distributed laboratory are no exception. However, in addition to these commonly known communication demands, distributed laboratory applications have several additional communication demands:

(1) *Data-driven, time-constrained computations* – in addition to satisfying the communication demands internal to a laboratory's parallel computational instruments, the communication infrastructure must also provide for suitable latencies of the data transfers from input engines to computational instruments, from instruments to observers' displays, and between multiple, collaborative displays.

(2) *Scalable communications* – because any number of different data generators and consumers may co-exist in a large-scale laboratory, it is not sufficient simply to transfer data, but the infrastructure must also permit suitable transformations on the data as it is being transferred, and it must provide mechanisms for efficient data distribution and collection among large numbers of consumers and providers.

As a result of these additional two demands, the distributed laboratory's communication infrastructure cannot rely on inefficient techniques like ASCII transmission of data to handle

such issues as heterogeneity in the execution environment. The infrastructure must also facilitate the association of computations with data transfers, as explained in more detail in Section 2.2.4.

2.2.2 Operation in a Diverse Environment

The application environment described above is clearly heterogeneous, but it can perhaps be more strongly described as *diverse*. This means that, in addition to operating in an environment where machine-level representations of elementary types may differ, the various components of the application may be developed in a much less tightly coupled manner than a traditional parallel or distributed program. That is, they may not all be produced by a single group or organization and they may include "generic" components designed to operate in many situations.

One implication of this application diversity is that it is not generally practical for all of the components operating in the distributed laboratory to agree at compile-time on the formats of all data records to be exchanged during execution. Even if agreeing on a common and consistent format for the data were possible, systems depending upon such agreement are very fragile and unsuitable for a dynamic environment. This is a problem that has not been addressed in the past by high-performance communication systems. The most obvious approach to operating in an environment this diverse is to transmit data in some easily-interpreted free-form manner, such as encoded in ASCII, and to parse it upon receipt. This approach achieves the required flexibility, but at the cost of seriously compromising the efficiency of the communication system, and so it is not an option for a distributed laboratory.

2.2.3 Dynamic Connection of Clients and Data Flow Management

In the distributed laboratory environment, computational agents may be very long-running, and physical instruments may always have data available. In this environment it is natural to expect displays, their associated data processing agents and other components of the execution environment to come and go as computations proceed. To accomplish this, external clients must be able to locate and connect to data sources dynamically. This functionality tends to be lacking in communications tools created with relatively static topologies in mind (*e.g.*, MPI). Even those which have dynamic client creation may not support external connections or have adequate client/server location facilities (*e.g.*, PVM).

The potential for dynamic connection of clients also means that the flow of data and the processing needs experienced by these flows in the distributed laboratory are themselves very dynamic. In particular, it implies that senders may not have *a priori* knowledge of the identity of all possible receivers. This is a problem for communications systems which require destinations to be explicitly specified by the sender, such as traditional send/receive and RPC-style communication mechanisms. One can, of course, program the application such that it responds to the dynamic connection of clients and automatically routes information flows to interested parties. However, such implementations tend to make inefficient use of network resources[19] and to require the application programmer to create some complex code

on top of the basic one-to-one communication mechanisms. Using a generative, or publish-subscribe, communications model instead hides the complexity inside the communications layer and gives that layer more flexibility exploit the capabilities of the underlying network.

2.2.4 Information Filtering and Analysis

As already stated above, a communication infrastructure for distributed laboratories must manage both data flows and the necessary computations controlling those flows and their content. The resulting richness and dynamic nature of the environment and its traffic create additional opportunities for optimization and improvement. For example, a capable communications system may take advantage of external knowledge of the application and reduce overall communication requirements through the use of application-specific filtering and analysis. Because of the application-specific nature of the computation, the main contribution a communication infrastructure can make is in easing the construction of filters and computational agents and providing simple mechanisms through which application-level routines can be activated.

3 Contributions of DataExchange

The DataExchange library addresses the requirements described in Section 2. First, DataExchange is based on PBIO[4], a binary I/O package which provides a variety of mechanisms for handling operation in a diverse environment. Second, DataExchange provides facilities for naming and locating data sources, for dynamically connecting to those data sources and for automatic and configurable redirection of data flow. Third, to support flexible data processing, analysis and reduction, DataExchange also offers an active-message-style processing message processing facility. This processing facility can also serve to further refine and extend DataExchanges data flow management functionality by making content-based message forwarding decisions. The remainder of this section will briefly examine these features of DataExchange and, where applicable, compare them with the features of similar packages.

3.1 Binary I/O Support

Most modern communications libraries provide at least some support for transparently transferring information between machines with differing data representations, but the extent of support varies. PVM[10] allows per-message construction and transfer of information through packing and unpacking of elementary data elements in communication buffers. The application is responsible for ensuring that unpack operations occur in the same order as pack operations. MPI allows the specification and use of user-defined data types in communication. User-defined types in MPI can be defined recursively to create structures which include elementary and user-defined data types and arrays of those types. These user-defined types may be used directly in send and receive operations and the MPI library will perform any necessary packing and unpacking operations. In MPI, type matching between sender and

receiver depends upon the type's signature, which is the sequence of elementary data types in the type definition and the application is responsible for ensuring that the types used by the sender and receiver match. Matching types must have the same sequence of field types, though they can be at different offsets in the record.

While the facilities for user-defined data transport provided by PVM and MPI suffice for the relatively tightly-coupled domain of parallel programming, they are less adequate for many cooperating programs that would make up a distributed laboratory. In particular, consider the problem of adding a new field to a widely used message format. Ideally, adding a new field to an existing message format should only affect those programs which must use or operate on that field. Programs which are not interested in the new field or which should merely forward its data without interpretation should not require modification. However, because both PVM and MPI require senders and receivers to agree on type signatures, adding a new field requires a simultaneous upgrade to all the programs in the lab that might operate on messages of the original format. This is obviously impractical in widespread and diversely managed environment like a distributed laboratory.

Similar problems arise in trying to create reusable distributed laboratory tools, the equivalent of instruments in a physical laboratory. One would like to have displays and filter programs that could be applied to a variety of programs and situations in the distributed lab without requiring *a priori* knowledge of the data formats involved. However, message passing systems which rely on implicit receiver/sender knowledge of the data formats being exchanged do not generally transmit sufficient information for an external party to meaningfully interpret a message in an unknown format.

To address these types of issues, DataExchange relies on PBIO to support its data transfer operations. In addition to other features, such as named data types and the ability perform type conversion between matching fields of different basic types, PBIO supports more elaborate and flexible type matching than either PVM or MPI. In particular, PBIO supports type matching based on *field names* rather than just elementary types. PBIO's type matching is also more flexible in that receivers are not required to specify every field in the types to be exchanged, but need only specify the names and types of the fields that they are interested in. Receivers will automatically extract the named fields of interest and ignore unspecified fields.

Name-based type matching while preserving efficient transmission is possible because PBIO transmits meta-level **format descriptions** before transferring any data records. A format description consists of the name of the record (message) format, and the names, sizes, offsets and data types of the each of the fields in the record. A previously registered format may also serve as the basic type of a field, allowing the creation of complex nested record types. PBIO also supports fields which are statically sized one or two dimensional arrays or dynamically sized one dimensional arrays of simple or derived datatypes. Record meta-information is transmitted only once. Thereafter, transmission occurs in the writer's native format and the PBIO library on the receiver transparently handles discrepancies between the writer's format and the format required by the reader. Transmitting in the writer's native format allows complex messages to be sent over the wire without any copy or "buffer building" stage. In the case of transfers between homogeneous machines, the only additional overhead

imposed by PBIO is the transmission of a 4-byte format ID.

Because PBIO record meta-information is transmitted on the wire rather than relying on implicit knowledge in the sender and receiver, the task of creating generic tools and displays that can be "plugged in" to applications in the distributed laboratory is eased. Besides the ability to translate between wire and native record formats described in the previous paragraph, PBIO provides a variety of facilities for manipulating format meta-information and extracting individual fields. These functions support the special needs of programs which are required to process a portion of the data in a record and forward the remainder untouched.

3.2 Location, Connection and Flow Control

Section 2.2.3 describes the dynamic nature of the distributed laboratory environment. This nature creates the requirement that a new display be able to locate and connect to information sources and that communication topology be dynamic at run-time.

Dynamic location of communicators is seldom directly addressed by communication packages. Basic socket implementations require the use of a port number that is known to all parties and there are no common mechanisms for publishing a dynamically created name/port number association. PVM supports communication with a dynamically forked remote process, but lacks the ability to initiate communication between two unrelated processes. MPI contains no support for initiating communication between unrelated processes, though MPI-2 will provide some primitive communicator location facilities. CORBA[16] recognizes the problem of dynamic location and includes a specification for the CORBA Name Service (which is getting widespread implementation). The Spring Operating System[17] also has an extensive name service. However, Spring is not widely used, and the CORBA name service is not useful outside of the CORBA domain. While a naming service to support communication in the distributed laboratory need not be overly complex, it must be available.

The ability for a client to selectively receive only a portion of the data available from a particular source is also a capability that is seldom supported by communication libraries. However, the distributed laboratory environment has the potential for large flows of scientific and monitoring data. While clients always have the option of receiving everything and discarding that data which is not of interest to them, adding the ability for them to specify what is interesting so that the uninteresting is not transmitted (and so does not take bandwidth) may be key to reducing overall network resource demands.

DataExchange contains facilities to support the location, connection and dataflow control needs of a dynamic communication environment. Its simple name server helps clients locate and connect to data sources in the distributed laboratory. DataExchange also allows clients and data sources to manage the redistribution and transmission of data on a per-message-type basis. This allows DataExchange to support highly dynamic applications while limiting unnecessary network overhead. These facilities are explained in greater detail in Section 4.

3.3 Information Processing and Filtering

In an environment like the distributed laboratory, it is useful to be able to create generic filter and display programs that can be assembled in a plug-and-play manner. Both filter and display-type programs are often constructed in an event-driven programming style. Like a variety of other packages, DataExchange provides the ability to bind application level functions to communication system events. DataExchange events include the connection of new clients, shutdown of client connections, and message arrivals. DataExchange also allows different functions to be bound to each type of message. With this level of support, simple filter programs can often be reduced to a very few lines of code.

This basic support for binding application functions to message arrivals also provides for the creation of data-dependent routing and forwarding of messages. Data-dependent routing could be used to supplement the record-type-based forwarding that is built in to DataExchange.

4 The DataExchange Communication Library

DataExchange facilitates communication between processes in a distributed laboratories environment through a set of services that provide the behavior needed by applications. For example, a visualization client receiving large amounts of data from a global atmospheric model may at some point no longer need messages containing updated species concentration values. Instead of being forced to discard the species concentration messages upon arrival, DataExchange provides a means for the client to issue a command to the server to stop sending messages of that kind, thus freeing bandwidth that would otherwise be used. DataExchange currently executes on Sun Solaris, SGI, Linux, RS6000, and Windows NT platforms. A detailed description of the library interface can be found in [6].

4.1 Concepts

Before discussing the services provided by DataExchange, we introduce several general concepts, an understanding of which is useful for understanding those services. One enables Dataexchange communication by creating a *DataExchange context*. The context, specific to a single process, supports a group of related connections. The separate communication context insulates communication internal to the library from external communication and provides a means of associating related communication channels. A DataExchange context is created with a call to `DExchange_create()`, which creates the context by initializing a `DExchange` data structure. The returned pointer to the data structure is attached to subsequent calls to the library.

An individual connection has associated with it a *DEPort structure*. The DEPort structure maintains information about the connection, such as its hostname and port id. A DEPort structure is created automatically when a connection is established.

As described in Section 3.1, efficient transmission is possible because PBIO uses meta-level *format descriptions*. Because format descriptions may differ slightly due to byte ordering and word size differences between platforms, DataExchange has mechanisms to resolve differences in format descriptions when encountered. This resolution takes place only once within the context of the DataExchange with which the format was registered. That is, a format description received within one DataExchange context will not invoke format resolution against a format description received within another DataExchange context. An application registers its format descriptions within its local DataExchange context. The DataExchange then forwards the format registration to all connections within its context. This forwarding step is necessary to ensure that data gets forwarded to all processes requesting such data. Formats are registered using `DExchange_register_format()`. Each format is assigned a format ID which is used to identify the data type on subsequent operations.

4.2 DataExchange Services

4.2.1 Dynamically Locating and Connecting to Data Sources

In a distributed laboratories environment it is natural to expect displays, their associated data processing agents, and other components of the execution environment to come and go over the course of time. A library that supports dynamic locating and connecting to data sources must provide several services: dynamic location of communicators, dynamic connection to the communicator, and the ability to direct events to the new client without disrupting the data source.

Dynamic location of communicators in an environment characterized by a non-static topology is seldom addressed by communication packages. DataExchange provides a means for locating communicators in such an environment through a *group server*. The group server is a simple name service that relieves an application of the responsibility of knowing the specific location <hostname, port> of a connection point. The application need only know an agreed upon name to locate and connect to a connection point. The group server implements its service as a repository of name, connection pairs. The name, a <user name, application name, group type> triple, uniquely identifies a connection point. *User name* is usually the userid associated with the process establishing the group. If not given, it is derived from the execution environment. The *application name* is an agreed upon name identifying the set of communicating processes. *Group type* is provided to further delimit the application name. Group type is used in cases where an application supports several connection points for different purposes. For example, one of the processes participating in the global atmospheric simulation may establish a connection point for all distributed components of the simulation. That same process may need to establish a communication point to enable communication between the model and the filters and visualization/steering clients. The two communication 'groups' would be established with the same application name but differentiated by group type.

The group server accepts queries based on any field of the <user name, application name, group type> triple, and supports both exact matching and regular expressions. In response

to a query, the group server returns a list of connection points from which the process originating the query can select.

Dynamic connection to a data source requires the ability of an event-based application to respond to arriving events, including connection request events. Event-based processing is supported by the `DExchange_poll_and_handle(DExchange de, int block)` routine, the basic event handling call in DataExchange. The poll and handle function is analogous to the `XtAppNextEvent()/XtAppDispatchEvent()` pair in X windows toolkit programming where a DataExchange event is a basic communication occurrence, such as a connection request or message arrival. The function queries all communication channels associated with a DExchange context and processes events from each connection having input ready. The `block` parameter specifies whether or not DataExchange should block (suspend program execution) waiting on an event if none is ready.

In addition to services for dynamically locating data sources and accepting connection requests dynamically, a communication library meeting the dynamic needs of distributed laboratories must also provide a means for a newly connected client to automatically begin receiving data for which it has registered an interest. DataExchange provides such a service through default forwarding behavior. A DataExchange server will automatically forward all messages it receives to all connections that have registered an interest in the messages. This default behavior can be modified through control functions described in Section 4.2.3 that allow the selective forwarding and blocking of messages based on format description.

4.2.2 Analyzing and Filtering Data

Many event-based applications must handle arriving messages of multiple data formats where often the action to be taken is specific to the data format. DataExchange provides a means for associating an action to a data format through **action routines**. Action routines are routines associated with a particular data format and invoked when a message of that format arrives. Similarly, applications may register routines to be called when a connection is established or shutdown. Action routines may be registered and unregistered with DataExchange at any time.

DataExchange provides two types of action routines for handling incoming data, functions and filters. **Functions** are called *after* any implicit data forwarding whereas **filters** are called *before* data forwarding and can return a value which preempts further processing. Functions are useful for gathering statistical or debugging information on the event stream. Filters, on the other hand, are useful for filtering events, reordering an event stream, or consuming events and generating derived events. Functions and filters are associated with specific data formats though the same subroutine can be registered as a handler for multiple data formats.

4.2.3 Controlling Flow of Data

As mentioned previously, the default DataExchange behavior is to resend all data received from one connection to all of its other connections (except the sender). This mechanism is useful for constructing interesting servers and managing data flow in networks of communi-

cating DataExchange programs. The DataExchange library allows control of this resending in ways which can be loosely divided into *record forwarding* and *record blocking* controls. Record forwarding and blocking controls both affect the implicit resending of received data, but in different ways. Forwarding controls are available to the DataExchange program performing the resending operation and allow that resending to be controlled on a per-format basis. Forwarding controls are therefore local controls in the sense that a DataExchange program calling those routines affects its own resending. The complementary record blocking controls are instead used by the DataExchange programs *to whom data is being resent* to affect the type of data that is resent to them. In particular, by using the record blocking controls a DataExchange program can tell another communicating program *not* to send it records of a particular format. Record blocking controls thus affect resending on a per-connection basis, but they differ from the forwarding controls in that blocking routines invoked by one DataExchange program affects resending that occurs *in other communicating programs*. In this sense, record blocking is a remote operation and is useful to any DataExchange program, be it client or server, that wants to influence what data is sent to it by others to whom it is connected. On the other hand, record forwarding controls affect resending in the DataExchange program invoking the controls and are thus a local operation. In both record forwarding and blocking, DataExchange's decision to send or not send a particular record to a particular client is based solely on the type or format of the record.

4.3 Sample Application

To understand the utility of DataExchange's features, it is useful to examine a sample application. In this paper we examine the global atmospheric model mentioned previously. This application is a collaborative effort between computer scientists and atmospheric scientists at Georgia Tech. Models like these are important tools for answering scientific questions concerning the stratospheric-tropospheric exchange mechanism or the distribution of species such as chlorofluorocarbons (CFCs), hydrochlorofluorocarbon (HCFCs) and ozone. Details of the model's solution approach, parallelization, and performance results are described in [11].

A common scenario in a distributed laboratories environment is to connect one or more visualization/steering/filtering/analysis clients to such a scientific model. We base our example on this scenario. Our simple example consists of three applications: a supplier, a filter/forwarder, and a consumer. The *supplier* (i.e., model) generates species concentration data (species concentration is the concentration of a chemical species such as ozone over an area) that is sent to the clients. Generating actual species concentration data requires executing the model. To keep our example simple and focused on DataExchange communication mechanisms, we hide the complexity of data generation by obtaining data through a call to a procedure where the procedure is left unspecified. The second application is a *filter/forwarder* that accepts connections from the supplier and visualization clients then filters records it receives based on a user supplied range of levels. The final component, the *consumer* emulates the communication behavior of the visualization/steering client. It accepts filtered records for processing but as was done for the supplier, the processing is

```
typedef struct _Limits {
    int lower_bound;
    int upper_bound;
} Limits;

typedef struct _XMix {
    int numSteps;                   /* logical time step             */
    int level;                      /* atmospheric level:  0..36     */
    int count;
    double * mix_ratio; /* species concentration in spectral form  */
} XMix, *XMix_ptr;

static IOField x_mixing_list[] = {
    {"numSteps", "integer", sizeof(int), IOOffset(XMix_ptr, numSteps)},
    {"level", "integer", sizeof(int), IOOffset(XMix_ptr, level)},
    {"count", "integer", sizeof(int), IOOffset(XMix_ptr, count)},
    {"mix_ratio", "float[count]", sizeof(double), IOOffset(XMix_ptr, mix_ratio)},
    {NULL, NULL, 0, 0}
};
```

Fig. 2. Types and format descriptions used in example

left unspecified. Processing the actual client would include transforming the data from its spectral form used in the model to a grid form needed to make the data useful for display using a 3D graphics library such as SGI Open Inventor.

Figure 2 contains type definitions and format definitions used throughout the example. The first type definition is of a client data structure used by the filter/forwarder application. The second type definition defines a single species concentration record. It's format description for purposes of binary encoding/decoding is given as the static array `x_mixing_list`. The `XMix` type and `x_mixing_list` format description are used in the supplier, forwarder, and consumer processes.

The supplier, such as is shown in Figure 3, creates a DataExchange context with a call to `DExchange_create()` and registers a data format, *x mixing ratio*. Registering the format has the effect of telling DataExchange how to interpret the buffer it receives when a subsequent calls to `DEport_write_data()` are made. The supplier then queries the group server with the application name, group type pair <"atmosphericModel", "model"> which returns a list of connection point(s) matching the query. Connection establishment is accomplished with `DExchange_initiate_first()` where the first parameter is the DataExchange context to which the connection will belong. The second parameter, `group_return`, is the structure returned by the group server containing a list of connections satisfying the query. `DExchange_initiate_first()` will traverse the list, attempting to make a connection with each connection point; it will return upon making the first connection. The final parameter is the block_by_default parameter. The value of FALSE tells the remote server, the forwarder in this case, that it should not block delivery of data to this client. The return value is a DEPort value that is used in subsequent reads/writes. The model then obtains the ID for the format it previously registered before dropping into a loop that generates and sends data.

```
void main()
{
    XMix * xmix;
    DEPort dep;
    comm_group_return group_return;
    int format_id;
    DExchange de;
    de = DExchange_create();
    DExchange_register_format(de, "x mixing ratio", x_mixing_list);
    group_return = matching_comm_groups("atmosphericModel","model" );
    if (group_return->count > 0)
        dep = DExchange_initiate_first(de, group_return, FALSE);
    format_id = DEget_format_id (de, "x mixing ratio");
    while (1) {
        xmix = gather_x_mixing_data();
        DEport_write_data (dep, format_id, (void *) xmix);
    }
}
```

Fig. 3. Supplier of Species Concentration Data

```
static int xMixFilter(DExchange de, DEPort dep, int format_id,
                      void *data, int data_length, void * cli_data_arg)
{
    XMix *rec = (XMix *) data;
    Limits *lim = (Limits *) cli_data_arg;
        /* return value controls forwarding */
    return !(rec->level < lim->lower_bound || rec->level > lim->upper_bound);
}

void main(int argc, char *argv[])
{
    char *group_id = NULL;
    Limits *cli_dat = malloc(sizeof(Limits));
    DExchange de = DExchange_create();
    cli_dat.lower_bound = atoi(argv[1]); /* lower and upper bounds for */
    cli_dat.upper_bound = atoi(argv[2]); /* filtering of atmospheric levels */
    DExchange_register_format(de, "x mixing ratio", x_mixing_list);
    DExchange_register_filter(de, "x mixing ratio", xMixFilter, (void*)cli_dat);
    if (DExchange_listen(de, 0) == -1) exit(-1);  /* open socket, select port */
    group_id = setup_comm_group(NULL, "atmosphericModel", "model", 300,
                                DExchange_host_name(de), DExchange_inet_port(de));
    while (1) {
        DExchange_poll_and_handle(de, TRUE);
    }
}
```

Fig. 4. Forwarder that filters species concentrations depending upon a user specified range of levels.

```
int numSteps;                          /* current logical time step */
static void xMixHandler(DExchange de, DEPort dep, int format_id,
                        void * data, int data_length, void * cli_data_arg)
{
    XMix * rec = (XMix *) data;
    numSteps = rec->numSteps;
    process_species_concentration(rec);    /* do some work */
}

void main()
{
    DEPort dep;
    comm_group_return group_return;
    DExchange de = DExchange_create();
    DExchange_register_format(de, "x mixing ratio", x_mixing_list);
    DExchange_register_function(de, "x mixing ratio", xMixHandler, NULL);
    group_return = matching_comm_groups("atmosphericModel", "model");
    if (group_return->count > 0)
        dep = DExchange_initiate_first(de, group_return, FALSE);
    while (1) {
        DExchange_poll_and_handle (de, TRUE);
        if (numSteps = 5000)
            DEport_set_format_block(dep, "x mixing ratio", TRUE);
    }
}
```

Fig. 5. Consumer accepts filtered species concentration messages until logical time reaches 5000.

The forwarder, as shown in Figure 4, creates a DataExchange context, registers the format *x mixing ratio* and the action routine `xMixFilter`. `xMixFilter` is a handler routine invoked upon receipt of messages of type *x mixing ratio*. Once the action routine is registered, the forwarder establishes a group with the group server with a call to `setup_comm_group()`. The group is established with the name <"atmosphericModel", "model">. The final parameter, the integer 300, specifies the length of time in seconds for which the group will remain active. Once the group is established and the clients begun, the forwarder accepts events, invoking the handler which filters species concentration records for levels outside the range specified by the user, then forwards qualifying records to all connections except the one on which it received the message.

The final application component, shown in Figure 5, is the consumer. The consumer registers the format *x mixing ratio* and the action routine `xMixHandler`. Though the format registered is the same as is used by the supplier and filter/forwarder, it need not be. DataExchange handles differences not only in byte order but also in field ordering and count as well. `xMixHandler` handles species concentration messages (of data type *x mixing ratio*). The client then queries the group server and initiates communication with the filter/forwarder in the same manner as was done by the sender. The client then drops into a while loop, servicing events. An arriving message automatically invokes the handler to process the message. To illustrate blocking, we have included the *if* statement following `DExchange_poll_and_handle`.

When logical timestep 5000 is reached, `DEport_set_format_block()` is called with the format *x mixing ratio*. The effect of the call is to block further transmission of species concentration messages at the filter/forwarder.

5 Performance

To a great extent the performance of DataExchange depends more strongly upon the performance of the underlying network transport layer than on any characteristic of DataExchange itself. Some DataExchange features, such as the ability to block the transmission of unnecessary data, may yield performance gains over approaches that do not have such support. However, the extent of the performance gain depends strongly upon the application and the amount of unnecessary data that DataExchange prevents from being transmitted. Since a particularly well-crafted application would transmit no unnecessary data, the strength of DataExchange would be that it facilitates the construction of plug-and-play applications that are well-crafted in that respect.

However, a unique aspect of DataExchange that does affect performance is its support for application diversity. That is, DataExchange is able to support the rapid exchange of binary data between application components without *a priori* agreement on the fields being exchanged, much less the types, sizes and layout of those fields. Because PBIO always transmits information using the native data layout of the sender,[1] this implies that a *format conversion* must be performed on the receiving end before the message can be processed as native data. When identical structures are exchanged between homogeneous machines, no conversion is necessary. But some conversion processing is required when data is exchanged between application components whose data representations vary in the number, types or layout of fields in the data, or if the communicating machines have differing native data representations, such as using different byte orders. Furthermore, because DataExchange has no *a priori* knowledge of the native formats used by the program components it might communicate with, the precise nature of this format conversion is dynamic at run-time. This conversion is another form of the "marshaling problem" that occurs widely in network communication. That marshaling can be a significant overhead is well known[3] and tools such as USC[13] attempt to optimize marshaling with compile-time solutions. Unfortunately, the dynamic form of the marshaling problem in DataExchange rules out such static solutions. As described above, the conversion overhead is nil for some data exchanges, but for others, it can be significant. To put the overhead in context, Figure 6 shows a comparison of the times required to transmit, copy and convert a message.

The particular values of given in Figure 6 depend upon a number of factors. The message transmit time is based on a 15Mb/sec transmission rate. The copy time is the time required to perform a bcopy on an UltraSparc 140. While both of the previous values vary strictly as a function of message size, the amount of time required to convert the message from the "wire" format to the receiver's native format is highly variable, depending upon the precise differences between the two formats. In this case, the values are derived from a situation where

[1] As described in Section 3.1.

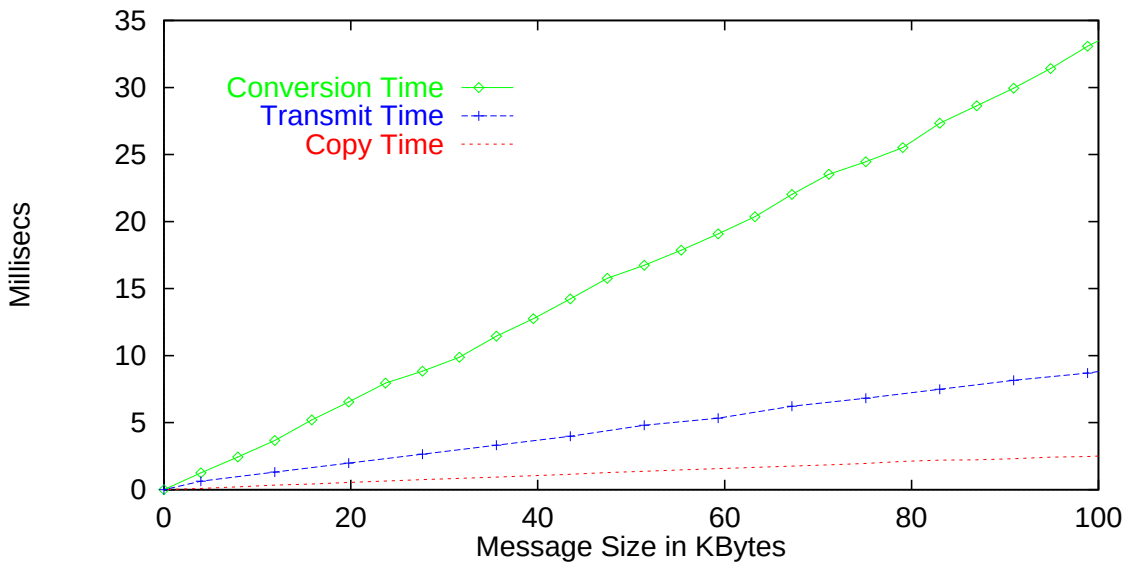

Fig. 6. Comparison of convert, copy and transmit times.

both byte-swapping and a size conversion for each data element were required to convert from wire to native format. While this represents a near worst case for data conversion times, the comparison is far from satisfactory. In this worst case, data conversion is taking three times as long as the time to transmit the message over the network and an order of magnitude longer than a simple move operation. In many situations, this might be an unacceptably high price to pay for supporting application diversity.

While a portion of the costs involved in doing the message conversion of Figure 6 are the simple consequence of the raw number of operations involved in performing the data conversion, a significant fraction of the overhead is due to the fact that the conversion is essentially being performed by an interpreter. This implementation choice and resulting excessive overhead is a direct consequence of DataExchange's lack of compile-time information about the precise nature of the conversion required. One would hope that a more efficient implementation would be able to bring the cost of conversion to close the the time required for the bcopy operation.

Since DataExchange is targeted towards high performance applications which can ill afford the increased communication costs associated with interpreted conversion, we have recently begun experimenting with dynamic code generation in an attempt to reduce these costs. The results, shown in Figure 7, have been highly encouraging.[2] For the type of conversion used in Figure 6, the dynamicly generated conversion routine operates between three and four times as fast as the interpreted version. This improvement removes conversion as the dominant cost in communication and brings conversion costs to much nearer the level of a copy operation.

However, employing dynamic code generation for conversions means that DataExchange must bear the cost of generating the code as well as executing it. Because of the format

[2] The dynamic code generation described in this paper was implemented using the Vcode package[7] developed at MIT by Dawson Engler.

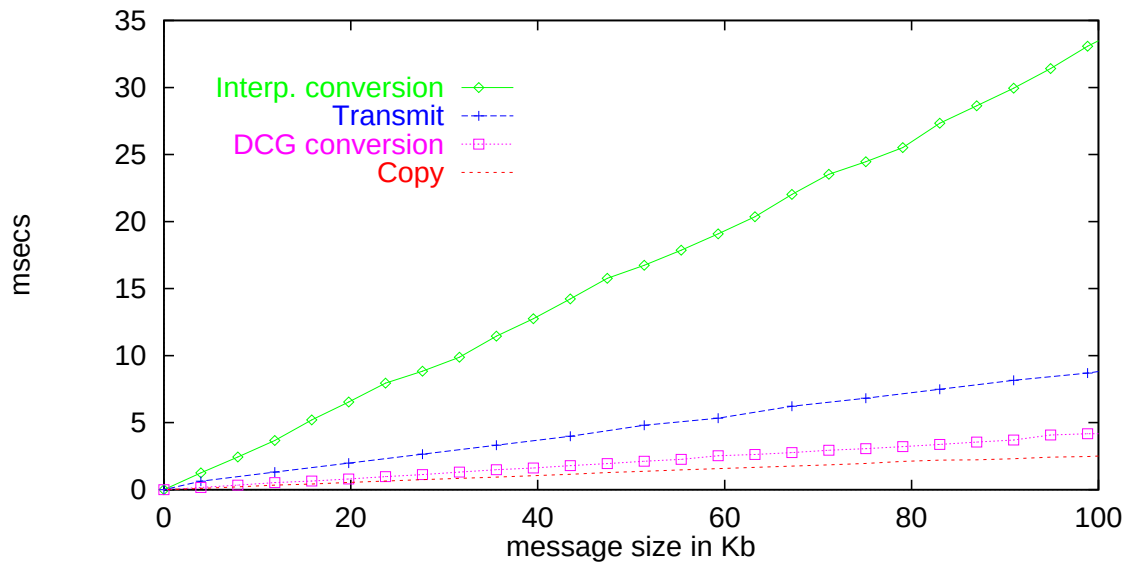

Fig. 7. Comparison of interpreted conversion, dynamically code generated conversion, copy and transmit times.

information in PBIO is transmitted only once on each connection and data tends to be transmitted many times, conversion generation is not normally a significant part of DataExchange overhead. Yet that overhead must still be considered to determine whether or not the use of dynamic code generation results in performance gain.

The proportional overhead imposed by doing dynamic code generation varies dramatically depending upon the internal structure of the record. This differs from the situation in Figure 7, where the worst-case conversion run-time is much more dependent upon the size of the message than its structure. To understand this variation, consider that the conversion code for a record which contains large internal arrays consists of a few `for` loops that process large amounts of data. In comparison, a record of similar size consisting solely of independent fields of atomic data types requires custom code for each field. The result is that for records which consist solely of arrays, dynamic code generation is almost always of performance benefit. For array-based records of around 200 bytes the time to generate and execute dynamic conversion code is less than the time to perform an interpreted conversion. At that point, dynamic code generation is a performance improvement even for formats that are only used once.

The situation is less clear for record formats consisting mostly of individual atomic fields. Figure 8 compares the times for such records. This figure shows that for this type of record dynamically generated conversion run nearly an order of magnitude faster than interpreted conversions, but the one-time cost of doing the code generation is relatively high. Obviously, if many records are exchanged, the costs will be amortized over the improved conversion times. But for one-time exchanges dynamic code generation for conversions may be more expensive than simple interpreted conversions.

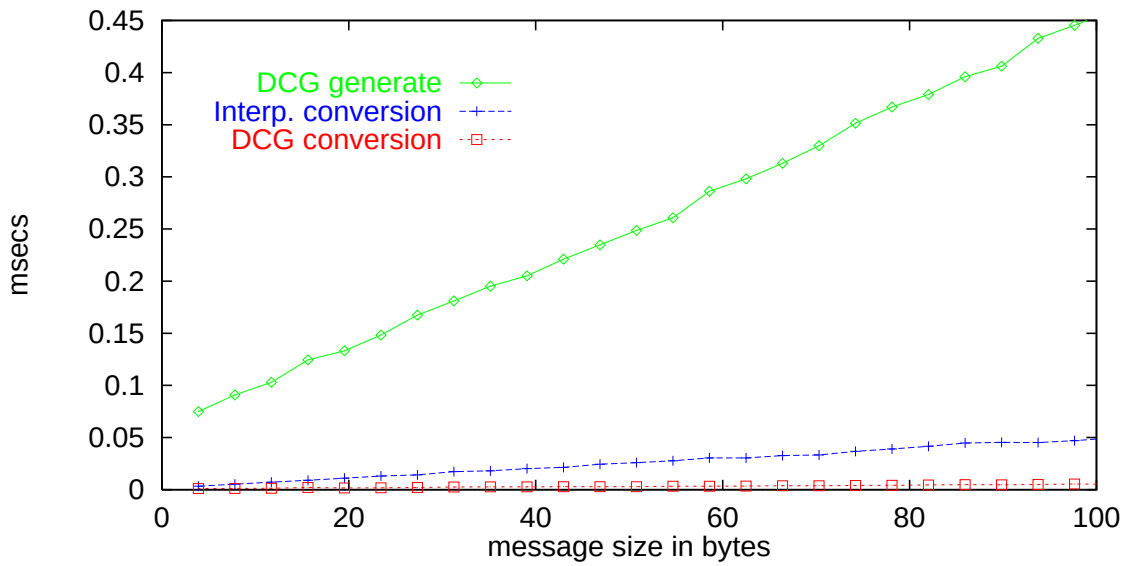

Fig. 8. Comparison of interpreted conversion, dynamically code generated conversion, DCG conversion generation times.

6 Conclusion

The current trends in high-performance computing applications are towards large loosely-coupled distributed computation and more dynamic models for both the computation itself and for run-time interaction with the computation. Unfortunately, these trends are not well supported by traditional message passing environments. This paper examines and abstracts the communication requirements for this class of applications, which are being studied in the Georgia Tech Distributed Laboratories Project. We use these requirements to motivate the discussion of DataExchange, a communications infrastructure developed to support Distributed Laboratories. DataExchange has proven to be a useful and flexible tool in creating complex plug-and-play interactive applications, including a global climate model, abstracted here as a sample application. We also include results on a novel use of dynamic code generation within the communication library. These results impact DataExchange performance in its support for application diversity, a less restrictive approach to programming cooperative programs that removes the requirement for *a priori* agreement on the contents and formats of information to be exchanged within the application. We show that application diversity can be supported without great impact on communication costs.

References

[1] Scott B. Baden. Software infrastructure for non-uniform scientific computations on parallel processors. *Applied Computing Review*, May 1996. ACM Special Interest Group on Applied Computing.

[2] Ken Calvert, Robin Kravets, and Boby Krupczak. An extensible end-to-end protocol and framework. In *Proceedings of the 1995 IFIP International Working Conference on Upper Layers Protocols, Architectures and Applications (ULPAA)*, December 1995.

[3] D.D.Clark and D.L.Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Synposium*, pages 200–208, Sept 1990.

[4] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *(anon. ftp from ftp.cc.gatech.edu)*.

[5] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. Falcon – toward interactive parallel programs: The on-line steering of a molecular dynamics application. In *Proceedings of The Third International Symposium on High-Performance Distributed Computing (HPDC-3)*. IEEE, IEEE Computer Society, August 1994. Also available as Georgia Institute of Technology Tech Report GIT-CC-94-08.

[6] Greg Eisenhauer and Beth Schroeder. The DataExchange library. Technical Report GIT-CC-96-17, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, 1996. *(anon. ftp from ftp.cc.gatech.edu)*.

[7] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[8] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mcs.anl.gov/mpi/mpi2/mpi2.html.

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

[10] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[11] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

[12] LabSpace. Creating virtual laboratories for scientists. URL:http://www.cs.northeastern.edu/research/labspace.

[13] S. W. O'Malley, T. A. Proebsting, and A. B. Montz. Universal stub compiler. In *Proceedings of the SIGCOMM '94 Symposium*, Aug 1994.

[14] Scott Pakin, Mario Lauria, and Andre Chien. High performance messaging on workstations: Illinois fast messages (FM) for myrinet. In *Proceedings of Supercomputing '95*, December 1995.

[15] Beth Schroeder, Greg Eisenhauer, Karsten Schwan, Jeremy Heiner, Vernard Martin, and Jeffrey Vetter. From interactive applications to distributed laboratories. Technical Report GIT-CC-96-23, College of Computing, Georgia Institute of Technology, 1996. http://www.cc.gatech.edu/tech_reports.

[16] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., 1996.

[17] J. Stankovic and K. Ramamritham. The design of the spring kernel. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 146–157, December 1987.

[18] P. David Stotts and James M. Purtilo. Virtual environment architectures: Interoperability through software interconnection technology. In *Proceedings of the Third Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 221–224, Morgantown, W.Va., April 17-19 1994. IEEE Computer Society Press.

[19] Mark D. Wood and Bradford B. Glade. Information servers: A scaleable communication paradigm for wide area networks and the information superhighway. In *Proceedings of the 7th SIGOPS European Workshop*. Association of Computing Machinery, 1996.