Optimizing Remote File Access for Parallel and Distributed Network Applications

Jon B. Weissman*, Mahesh Marina+, and Michael Gingras+

Department of Computer Science and Engineering
University of Minnesota, Twin Cities*

Division of Computer Science
University of Texas at San Antonio+

(jon@cs.umn.edu)

Abstract

This paper presents a paradigm for remote file access called Smart File Objects (SFOs). The SFO is a realization of the ELFS (Extensible File Systems) concept of files as typed objects, but applied to wide-area networks [9]. The SFO is an object-oriented application-specific file access paradigm designed to address the bottleneck imposed by high latency, low bandwidth, unpredictable, and unreliable networks such as the current Internet. Newly emerging network applications such as multimedia, metacomputing, and collaboratories, will have different sensitivities to these network "features". These applications will require a more flexible file access mechanism than is provided by conventional distributed file systems. The SFO uses application and network information to adaptively prefetch and cache needed data in parallel with the execution of the application to mitigate the impact of the network. Preliminary results indicate that the SFO can provide substantial performance gains for network applications.[1]

1.0 Introduction

Network applications often require access to remote data sources. Unfortunately, the performance "features" of current wide-area network: high latency, low bandwidth, unpredictable latency and bandwidth, and poor reliability can be an obstacle for network applications. Collec-

1. This work was partially funded by grants NSF ACIR-9996418 and CDA-9633299, AFOSR-F49620-96-1-0472, and ARP 010115-226.

tively these network properties jeopardize the efficiency, predictability, and reliability of network applications. This is becoming a more pressing problem as there is an ever-increasing volume of interesting remote data sources available on the network including digital libraries, scientific databases, movies, and images.

In this paper, we consider two important classes of network applications, client-server and metacomputing. Client-server applications are typically characterized by a set of interacting components running at fixed locations in the network. This includes simple applications such as Web browsers accessing Web server files to more sophisticated collaboratories. Because many of the client-server applications are interactive, predictable network file access is an important objective. Metacomputing applications tend to be more dynamic with high-performance as the primary objective. This includes applications such as global climate modelling or space weather prediction in which remote data sources must be efficiently accessed throughout the computation. Both classes of applications are hampered by these network "features".

Remote file access can be performed in one of three ways: uploading, downloading, or remote access. Uploading moves the application to the file location and avoids many of the network bottlenecks. Unfortunately, this paradigm is limited to a small class of applications. For example, it would not be suitable for high-performance applications in which the file's location is unlikely to have sufficient computing resources. Downloading moves the file to the applications location. However, downloading is expensive both in time and disk resources, especially if the application does not require the entire file (e.g. browsing a multimedia file for content). Remote access allows the application to run in one location and access files remotely across the network. It is the most flexible paradigm in that it can support the widest variety of network applications including client-server and high-performance metacomputing applications. However, remote

access suffers from the performance and reliability properties of the network. What is needed is a flexible solution that can mitigate the impact of the network on the application.

We believe that the network bottlenecks must be addressed in an application-specific way because different applications have different sensitivities to network performance. For example, multimedia applications require predictable performance while metacomputing applications generally favor high-performance. Our solution is the Smart File Object (SFO), which is object-oriented *middleware* that sits on the "edge of the network" between the application and the remote file. It adopts the remote access paradigm for its flexibility and addresses the network bottlenecks by exploiting information about the application and the network to perform adaptive caching and prefetching. The SFO is based on the ELFS (Extensible File System) concept of files as typed objects to provide high-level interfaces and allow file specific properties to be exploited for performance [9]. SFOs provide a practical implementation of the ELFS ideas in dynamic wide-area networks.

The remainder of this paper is as follows. Section 2 presents related work. Section 3 describes the SFO architecture and prototype implementations. Section 4 presents three case studies and performance results for different applications and SFOs. Sections 5 and 6 present a summary and future work respectively.

2.0 Related Work

Related projects fall into the areas of wide-area distributed file systems and adaptive systems. Andrew is a scalable distributed file system that adopts client-side caching and session semantics to achieve efficient performance in a wide-area environment [12]. xFS is a "serverless" file system designed to remove the bottleneck inherent in server architectures such as in Andrew [1]. In

xFS, clients may cache and serve the file to other clients avoiding server interaction. Both Andrew and xFS are general-purpose distributed file systems designed to support efficient sharing of files in a wide-area distributed system. WebOS provides a URL-based mechanism for naming remote files and downloads the remote file upon access [13]. These systems each adopt the download model of file access which is not appropriate for all applications. The SFO paradigm allows application-specific file access policies to be easily implemented in user-space better meeting an application's performance objectives. For example, most if not all file systems, assume a sequential mode of file access and base caching decisions upon this assumption. However, this will perform poorly for files accessed non-sequentially. In contrast, the SFO caching mechanism can be tailored to the specific access pattern. The value of combining caching and prefetching for optimizing I/O performance in a global cluster-wide memory system is demonstrated in [14]. The authors present a very general approach that could most likely be applied to wide-area systems, but the paper focuses on fast system-area clusters.

RIO is a file access paradigm for remote parallel file systems and shares our goal of high performance [2]. The objective is to efficiently support parallel file system interfaces such as MPI-IO. The SFO is a more general concept that can support many different kinds of file types and interfaces including multimedia files, scientific DB files, parallel files, etc.

Adapting to network and application characteristics in an important property of the SFO paradigm. Adaptation is necessary when the network resources fluctuate in performance such as in a dynamic network environment. A number of research groups are exploring the concept of application adaptivity including Autopilot [11], and Active Harmony [8]. Autopilot is a toolkit that contains a library of runtime components needed to build adaptive applications. Active Harmony is exploring the concept of adaptability for threaded parallel applications. Madhyastha and Reed

have investigated adaptive file systems in which the I/O system learns and classifies the access patterns from a common set, and is able to optimize caching and prefetching in parallel systems given this information [10]. Within the SFO paradigm, the access patterns are specified via an API, which can be invoked at run-time to handle changing access patterns. In addition, our focus has been to apply optimizations to the problem of remote I/O in wide-area networks.

3.0 SFO Architecture

The SFO architecture has been designed with five goals: portability, flexibility, extensibility, usability, and high performance. Portability is important because SFOs will be deployed on a large variety of heterogeneous end-systems, networks, and metacomputing environments. The goal of portability is achieved by a *layered* implementation of *user-space* client-side modules that reside above the operating system and file system (Figure 1). Flexibility is needed because network applications have different sensitivities to network “features” and different objectives. Furthermore, it is not possible to fully anticipate the requirements of newly emerging network applications. This suggests that a flexible mechanism to support a more customized remote file access is needed. The decision to implement the SFO in user-space provides this important flexibility. Extensibility is another important goal since the varying requirements of new network applications may require the design of new SFOs. Extensibility means more than the ability to simply add new functionality to the I/O system (this is essentially met by our goal of flexibility).

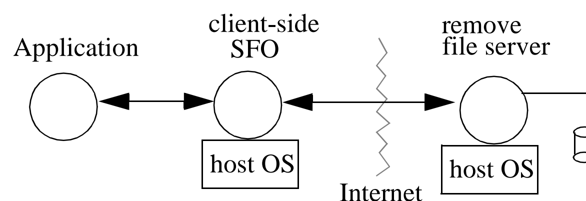

Figure 1: Layered architecture

It must be possible to extend existing I/O functionality since the task of building SFOs from scratch is likely to be time-consuming and difficult for some applications. Extensibility is met by adopting an *object-oriented* methodology for the construction of SFOs. Base class SFOs can be derived and extended to implement more complex SFOs. Usability refers to the ease of inserting and using SFOs in user applications. The object-oriented implementation of SFOs helps hide their internal complexity, while providing high-level file access APIs to user applications. Finally, the goal of high performance is met by adaptive caching and prefetching performed by the client-side SFO module. The client-side SFO module makes caching and prefetching decisions based on the pattern and frequency of data access from the application and the current network conditions. The SFO operates solely in an application-centric manner and adopts whatever sharing semantics are offered by the remote file system. It is fundamentally different from a traditional file system managed by the operating system designed to accommodate multiple simultaneous users. An important advantage of this architecture is that it does not require deployment of server-side SFO software which may not always be possible on some remote sites. In the remainder of this paper we describe the SFO architecture, and present results for three diverse applications that demonstrate the performance benefit of SFOs.

3.1 Extensible File Systems (ELFS)

SFOs are based on the concept of Extensible File Systems (ELFS) [9]. In ELFS, files are typed objects and class-specific information is exploited in order to optimize file layout, access, caching and prefetching. ELFS replaces the standard Unix file system abstraction as a sequential stream of bytes on disk. Instead, files are objects, and the application interacts directly with the file object for all file operations using high-level method invocation. ELFS file classes contain methods that represent the specific kind of file, and an interface that more naturally reflects how

the application would like to access the file. For example, an ELFS file class can be used to represent matrix files. In this file class, high-level methods such as `get_next_row`, `get_next_col`, etc. can be provided. The ELFS implementation transforms this high-level request into a sequence of lower-level I/O requests (e.g. a Unix `read`). In addition to improved interfaces, one of the key aspects of ELFS is the ability to exploit file specific and access specific information within the class. In an example given in [9], a 2-D matrix file that requires access by row and by column can be efficiently stored as a sequence of blocks on disk. ELFS classes are also used to represent parallel files in which file striping is transparent to the application. One of the primary objectives of ELFS is to attack the disk bottleneck for LAN-based parallel computing using such parallel I/O classes.

The SFO inherits two main ideas from ELFS: the use of file-specific classes to provide high-level interfaces and exploiting properties of the file class for performance. In this sense, the SFO may be viewed as an implementation of the ELFS conceptual ideas. However, the SFO differs in several respects: caching and prefetching is driven by predictive cost functions, it is adaptive to current network conditions, and the SFO is addressing the performance bottleneck in wide-area as opposed to local-area networks.

3.2 Inside the SFO

The SFO client-side module can interface with a variety of remote file servers and file systems, e.g. Andrew (AFS), Network File Service (NFS), Unix processes (UPS), Mentat File Objects (MFO), and Web Servers (WS), etc. (Figure 2). To date, we have implemented SFOs that interface with UPS, MFO, and WS remote servers as a proof-of-concept.

The SFO hides all details of the remote file from the application: in particular, its remote location and file server type. As an example, consider a matrix file SFO (Figure 3). This example

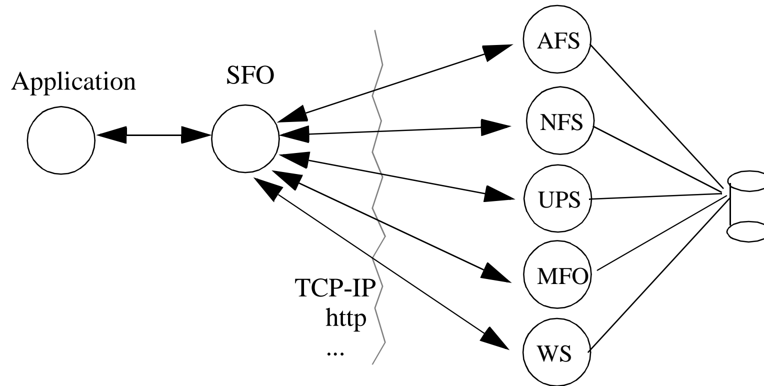

Figure 2: Client-side SFO

illustrates how a SFO can mitigate the network bottlenecks. While the application is computing with prior I/O data, the SFO can adaptively prefetch and cache the next set of needed data concurrently so that it is already on the "fast side" of the network, when the application next requests more data. In this example, the application is computing on a set of matrix rows while the SFO is prefetching additional rows in parallel needed later. The SFO has its own thread of control which allows concurrent execution. The SFO is programmed to be *application*- and *network-aware* taking into account current network performance and application demand in making prefetching decisions. The SFO calculates the amount of data to prefetch and cache after each application I/O request is served. This prefetch decision depends on the cost of performing remote I/O and the application rate of demand. Internally, the SFO monitors the current communication rate by timing the remote I/O operations to the file server, and the current application computation rate on each I/O data item by timing the interval between successive requests for data.

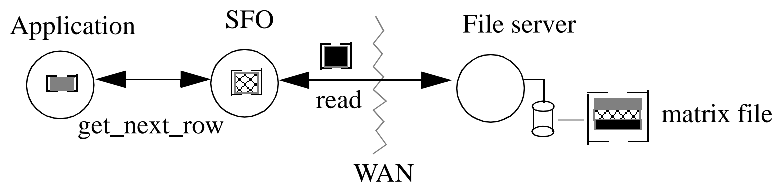

Figure 3: Matrix SFO in action

With this information the SFO can adapt its prefetching and caching strategies. This is particularly important for single-threaded SFOs such as in the Mentat implementation (MFO) because Mentat objects are single-threaded monitors[2]. For example, if the interval between successive calls to `get_next_row` is 200 ms, then the SFO may conclude that it has 200 ms to prefetch data. If the current communication rate was 100 ms per row, then the SFO could prefetch two rows. Prefetching additional rows could block the application since the SFO would be busy in network communications for > 200 ms. Prefetching fewer than two rows could result in a missed opportunity if the network bandwidth were to decrease later. For example, if the network communication rate later rises to 300 ms per row, then the remote I/O overhead could delay the application (300 ms > 200 ms). The SFO is adaptive to changes in network bandwidth and CPU performance. When the network bandwidth is higher, it can aggressively prefetch and cache data that could be used to avoid expensive network access later if the network bandwidth were to suddenly fall. Similarly, if the CPU load on the application machine(s) increases, then the requests for data from the application will slow down, giving a SFO a greater opportunity to prefetch. Later if the CPU load decreases and the application demand for data increases, cached data could be provided to avoid expensive network access. We expect machines to be shared in dynamic network environments, so machine load may indeed fluctuate during an application's execution.

Adaptive prefetching requires a prediction of the application computation time (or time between I/O requests) associated with each I/O data item (e.g. a row as in Figure 3) and the communication time to fetch a remote data item. We assume that the application performs identical computations on each I/O data item. This assumption means that the time taken to compute on each data item should be similar and vary only if the application machine load varies. The predic-

2. Multithreaded SFOs are the subject of future work.

tions of application computation time and communication are based on a weighted average of the recent past over a time window. More recent values appear to be a strong predictor [7][16][17] and an exponential decay function is used to model this relationship [15]. The SFO determines the application computation values, *comp*, by simply timing the interval between successive requests from the application. Similarly, the communication times, *comm*, are computed by timing requests to the remote file server. They are both scaled to give the time based on a single data item. The SFO also ignores "network spikes" in the prediction for communication since they appear to occur randomly. Fortunately, they also appear to occur infrequently. The following three cost equations are computed and used within the SFO:

$$T_{comm}[t] = \sum_{i=1}^{max(n,\,t-1)} \frac{(comm[t-i] \cdot 2^{n-i})}{2^n - 1} \cdot [1 - (comm[t-i] > kT_{comm}[t-1])] \qquad \text{(Eq.1)}$$

$$T_{comp}[t] = \sum_{i=1}^{max(n,\,t-1)} \frac{(comp[t-i] \cdot 2^{n-i})}{2^n - 1} \qquad \text{(Eq.2)}$$

$$N_{prefetch}[t] = max\left(1, \frac{T_{comp}[t]}{T_{comm}[t]}\right) \qquad \text{(Eq.3)}$$

The "time window" for prediction is n, where n is the prior number of I/O operations. The second term in the equation for T_{comm} is a boolean expression that has the value 0 or 1 and accomplishes spike removal. A spike is defined to be a communication time that is k times the prior predicted value. If a spike occurs, then the expression evaluates to 0 and that communication does not effect T_{comm}. If a spike occurs, we will go back to another prior value to ensure that T_{comm} is computed over the last n spike-free communications. The final equation gives the predicted number of data items to prefetch. Observe that it is adaptive to the computation and communication rates.

Because the SFO is programmed it can also implement much more flexible prefetching strategies than can OS-based file systems. The vast majority of OS file systems prefetch based on the assumption that the file will be accessed sequentially (and completely). This may not be true. In fact, the decision of "what to prefetch next" may be data- or end-user dependent. For example, the user may wish to move-back-and-forth through a multimedia file to browse for content. A SFO can easily handle this scenario and more arbitrary patterns of file access.

3.3 SFO Interface and Implementation

The SFO adopts an object-oriented interface that facilitates the construction of application-specific file classes following the ELFS paradigm. The object-oriented model allows different SFO classes to be easily defined and ultimately derived. The ability to derive SFO file classes is important because it means that SFO functionality need not be re-written for each and every application. For example, we can define a SFO matrix file class that provides basic file operations for matrices, and then later define a SFO class for sparse matrix files that inherits this interface together with the caching and prefetching machinery. We believe that all of this can be accomplished with minimal application changes in most cases. The application simply interfaces to the SFO for the desired I/O operations and the SFO can perform these activities without direct application involvement.

We illustrate the SFO with one of the prototype implementations in the Mentat system [4]. Mentat was selected as one of the prototypes because it supports an object-oriented model, objects are active with their own thread of control and communicate via non-blocking RPC, and it supports a global file system, the legion file system.[3] Active objects known as persistent Mentat

3. Mentat together with extensions for wide-area computing such as the legion file system were the prototype for the new Legion system [5].

objects are important because they allow the SFO to operate asynchronously and concurrently with the application. The non-blocking RPC allows non-blocking I/O to be easily supported. The legion global file system allows the SFO to access remote files through a convenient interface. The remote file is managed by a Mentat file object (not to be confused with the SFO) that runs on the remote system and acts as a file server to provide standard Unix-like file methods and semantics. Global files are prefixed by the special name `/legion`. The details of the legion file system can be found in [5].

The Matrix SFO of Figure 3 is easily implemented in Mentat using the Mentat Programming Language MPL (Figure 4). The `matrixSFO` class derives from a base SFO, called `basicSFO`, that provides basic file operations such as `open`, `close`, `get_file_size`, etc. The

```
class matrixSFO
{
  matrix_data *cache; // prefetched data
  enum {ROW, COL, BLOCK} read_type;
  read_type r;
  ...
public:
  matrixSFO(int,int,int);/row,col,elt sz
  ~matrixSFO ();
  // set type of read access
  void set_read_type (read_type);
  // read operations ...
  matrix_data *get_next_row ();
  matrix_data *get_next_block ();
  matrix_data *get_next_col ();
  set_interleave_factor(int);//default=1
  // write operations ...
    ...
};
```

```
persistent mentat class basicSFO
{
   file_object *remote_FO;
   // stored prior values of comp and comm
   // used for adaptive prefetching
   comp_data comp[MaxW],Tcomp[MaxW];
   comm_data comm[MaxW],Tcomm[MaxW];
    ...
public:
   int close();
   int open (string *, int, int);
   int get_file_size ();
   ...
};
```

Figure 4: MatrixSFO and BasicSFO interface. The MatrixSFO interface illustrates the high-level interface properties of ELFS.

basicSFO contains useful SFO infrastructure such as timers for computation and communication and an interface to the remote file server object, remote_FO. When open on the matrixSFO object is called (inherited from basicSFO), the remote_FO will be automatically created on the remote system. All subsequent I/O requests will be directed from the matrixSFO to the remote_FO which supports a standard Unix I/O interface, read, write, etc. Note that the application can indicate how it wishes to access the file (by row, column, or block, and with a particular interleave pattern) by calling set_read_type. This information is used by the SFO to decide what to prefetch. Implementation fragments for the SFO and for a client application are shown in Figure 5. Non-blocking I/O is supported in the application by delaying use of the return result until needed (Figure 5b).

In this simple example, get_next_row returns the next row that is in the cache, and then proceeds to prefetch the next set of rows. It is hoped that the next row is already in the cache, but

```
matrix_data *matrixSFO:: get_next_row () {
    matrix_data *data;
    char *buffer;
    int datasize, num_rows;
    // stop comp timer, set new value of comp
    ...
    // for simplicity, assume row in cache
    mentat_return cache; // RPC return value

    // now prefetch ...
    // determine num_rows via Tcomp, Tcomm
    datasize = num_rows * elt_size;
    // allocate data and buffer
    // start comm timer
    n = remote_FO-> read (buffer, datasize);
    // stop comm timer, set new value of comm

    // pack n and buffer into data
    add data to cache;
    // start comp timer again ...
}
```

(a) MatrixSFO implementation

```
main () {
    ...
    // create SFO for nxm matrix of int's
    matrixSFO mySFO (n, m, sizeof(int));
    mySFO.open
        ("/legion/m1.dat", O_RDONLY);
    for (i=0; i<n; i++) {
        // this is non-blocking!
        row = mySFO.get_next_row ();
        // do other work if we have any ...
        ...
        // now we need to use row -- this
        // will cause the system to block
        for (j=0; j<m; j++)
            sum += row[j] * vector[j];
        ...
}
```

(b) application fragment

Figure 5: MatrixSFO and application fragment

if it is not, the SFO will have to first prefetch it, blocking the application. This contingency case is not shown. The objective of prefetching is to avoid this situation. The determination of *num_rows* is where the SFO machinery will be used. Notice that the latest values of *comp* and *comm* are timed in this function. The *comp* time is the time between requests to this function, and the *comm* time is the time for the remote I/O. The application code fragment creates the SFO and then uses it to open a remote matrix file. The application code then loops requesting each row in a non-blocking fashion before using the row in a dot product calculation. This example is meant to be illustrative of how a SFO can be defined, implemented, and used, and has omitted some details for brevity.

4.0 Results

We have developed SFOs for three different applications: distributed matrix computations, parallel gene sequence comparison, and mpeg-view. The SFO within each application also interfaces with a different remote server type: Unix process, Mentat File Object, and a Web Server respectively. The range of applications and server types demonstrates the flexibility of the SFO approach and its wide applicability. Most importantly, however, the insertion of a SFO was able to boost performance for all of the applications. The results were obtained using an Internet testbed containing sites at the University of Virginia, University of Texas at San Antonio, and Southwest Research Institute in San Antonio.

4.1 Matrix Operations

Matrix operations are a simple example that nicely illustrates the performance benefits possible using SFOs for both reads **and** writes. The matrix SFO and client applications were written in C with TCP-IP communications. The remote file server is simply a Unix process that we have written to serve remote files to the client side of the network. A matrix SFO was created that sup-

ports methods similar to Figure 4. We have chosen to use matrix multiplication of square matrices A x B = C as a client application where A is stored with the client and B and C are remote matrices stored with the same server (Figure 6). We assume that B is stored in column-major order and it is read by column to enable an easy dot-product calculation. There are two SFOs used, one for B (reading the columns of B), and one for C (writing the columns of C).

In the first set of experiments, the application is sequential and running on a single UltraSparc. We compared the cost of different read methods for the SFO: download first, blocking remote access, static prefetch of a single column, and adaptive prefetch. Download first brings the entire matrix B to the application site before the application begins. Blocking remote access allows the SFO to fetch a column at a time on demand, but does not perform any prefetching. Static prefetch will always prefetch a single column while adaptive prefetch will adjust the amount of prefetching based on Eq. 3 using a window of n=5. We then compared the cost of different write methods: upload last and write overlap. In upload last, the SFO stores the entire result matrix on the client side and then transmits it to the server when it is completely formed. Write overlap allows the SFO to concurrently transmit the columns of C as they are computed in parallel with the application (Figure 7).

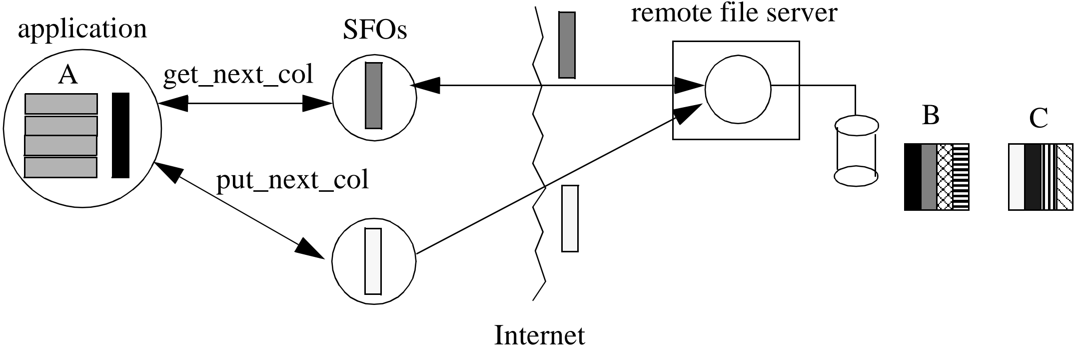

Figure 6: Matrix multiply using multiple SFOs

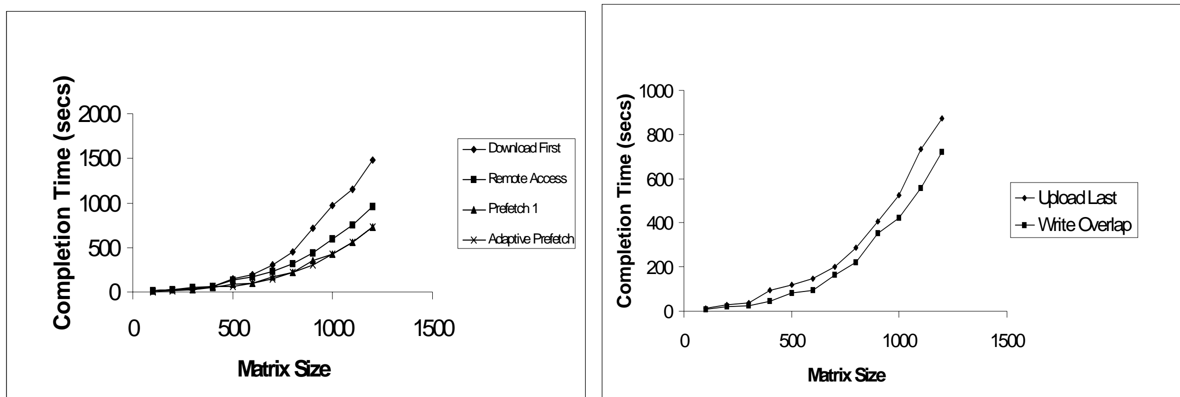

Figure 7: Sequential matrix multiply results. Application and remote file server are running on an UltraSparc. Each data point is the result of averaging at least five runs.

Download first is clearly the most expensive option. Remote access provides a 35% improvement by reducing network load and avoiding local disk access (the columns of B are stored only in the memory of the SFO). Static prefetch provides an additional 25% performance gain over remote access. For matrix multiply, adaptive prefetch provided only a modest performance gain since the Internet communication performance was fairly constant during the course of the runs and little opportunity for adaptation was available. However, these results indicate that the machinery employed to perform adaptive prefetching does not add significant overhead. In the next section, we will show an example where adaptivity improved performance due to Internet variance. The benefit of write overlap is around 20% (with read prefetching enabled).

In the next set of experiments, we use a parallel matrix multiply in which the rows of A are distributed in contiguous chunks across a set of processors (Linux cluster of x86 machines on 100 Mbps ethernet). The parallel application uses MPI for intra-application communication and TCP-IP for SFO communication as before. The processors are arranged in a pipeline where the head of the pipeline reads a column from the SFO and then it sends it down to the next processor and so on. However, each processor directly sends its computed result columns for C to the SFO for C.

For a 2Kx2K problem, the insertion of a SFO for reads and writes improved performance for a range of processor configurations (Figure 8).

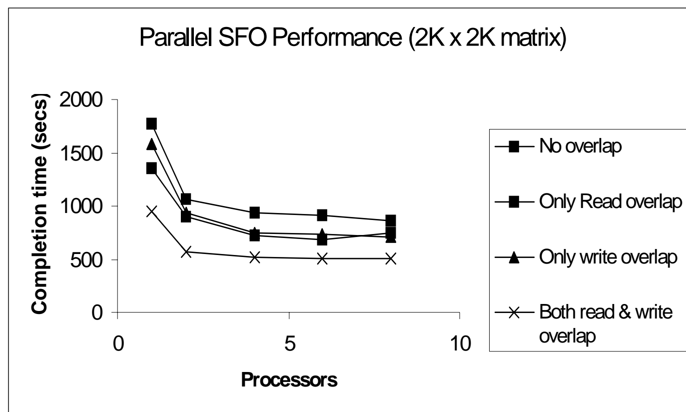

Figure 8: Parallel matrix multiply results. Application is running on a Linux cluster of x86 machines. Each data point is the result of averaging at least five runs.

4.2 Parallel Gene Sequence Comparison

Complib is a parallel Mentat application that compares a source library of DNA sequences to a target library of DNA sequences. The libraries are ascii text files. Complib utilizes several heuristics for string matching and we present results for *Fasta*, a well-known fast heuristic that sacrifices some accuracy for speed [6]. Complib statically decomposes the target library across a set of workers each assigned to a processor. The application then fetches a source sequence and passes it to all workers which compare it to their target sequences in parallel. This process repeats for each source sequence. The workers are arranged in a tree with the leaves performing the computation (Figure 9). The results contain a score for the current source sequence generated by each worker based on a comparison to its target sequences. Complib is a computationally-intensive application and is well-suited to metacomputing since it is likely that the data sources (source and target libraries) may in fact, be geographically separated from the actual computation (the workers).

We have modified Complib to experiment with SFOs (Figure 10). The source library is located remotely via the `/legion` file system with access controlled by a SFO, `SourceSFO`. The basic

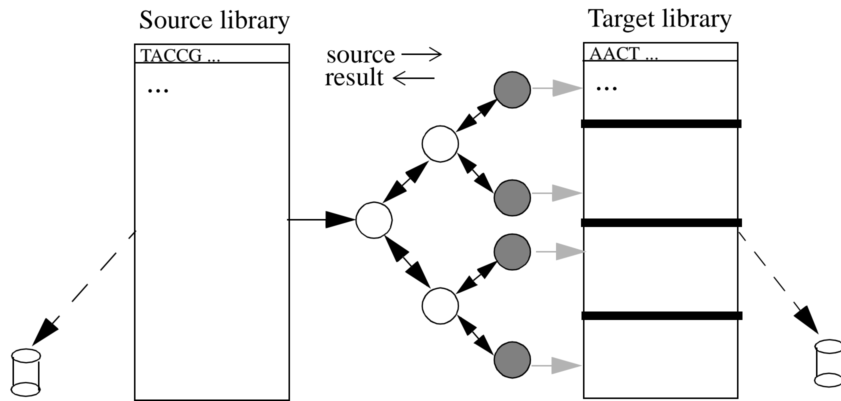

Figure 9: Complib structure

I/O data item in this application is a source sequence (4K bytes), and the *comp* and *comm* functions of (Eqs. 1-3) are based on a single source sequence. The target library and the workers were collocated in the same site. The job of the `SourceSFO` is to prefetch source sequences in parallel with the sequence comparison computation of the workers. The primary operation supported by the `SourceSFO` is `get_next_sequence` which delivers the next source sequence to the application. The `SourceSFO` can access the remote source library in the same four modes as in the matrix application: download first, blocking remote access, static prefetch 1 sequence, and adaptive prefetch. The download first option was implemented by the developers of Complib – it first downloads the entire source library file first without any prefetching or overlapping. The next option keeps the file remote and returns a single sequence at a time without prefetching. The third option uses the "generic" property of a SFO to prefetch needed data – the next sequence is prefetched while the current sequence undergoes the comparison computation. The adaptive prefetch option adjusts the rate of prefetching based on the current network and load conditions as governed by (Eqs. 1-3) to further improve performance. We compare these methods and show that the SFO machinery actually works and performance is improved for Complib.

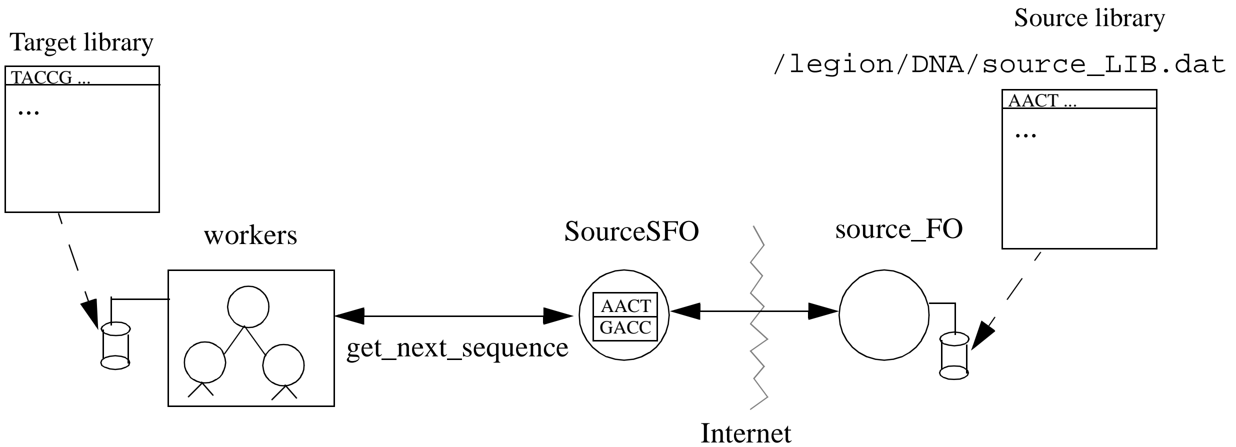

Figure 10: Complib with SFO. The application, SourceSFO, and target library are all located on the "fast" side of the network. The source library file is located remotely in `/legion` and managed by source_FO. Access to the source library file is controlled by the SourceSFO which adapts its behavior to the network and application.

We performed a series of Complib experiments using a fixed source library of 100 sequences and several target libraries: TG-12, TG-25, TG-50, and TG-100 (12, 25, 50, and 100 sequences respectively) that show the spectrum of performance for the SFO. Each sequence in the target and the source library was 4K bytes in length. The results are summarized in Table 1.

As expected, downloading is the most expensive option (as performed in the original Complib). Remote access of a single sequence performs better than downloading, and prefetching is clearly a performance advantage for this application. The precise benefit of prefetching depends on the particular problem instance and network. Since the application requests a single sequence one at a time, prefetching a single sequence is sufficient in the event that communication and computation performance is constant. However, in the event that communication and/or computation performance exhibits variability, the SFO can adapt to further improve performance. For the Complib runs, we observed that Internet communication performance was not constant [15]. We ran the `SourceSFO` with a "time window" of n=5 and n=10 to see if this variance could be exploited. The results indicate that performance could be further improved for appropriate prob-

Problem instance	Download first	Remote Access	Prefetch 1	Adaptive Prefetch (n=5)	Adaptive Prefetch (n=10)
TG-12	18728	17990	16489	**16091**	16265
TG-25	21821	20854	17148	17034	**16419**
TG-50	26321	23863	17070	17242	**16547**
TG-100	37466	31959	27205	25994	**25947**

Table 1: Complib results. All times shown are in milliseconds. The data was collected over a two hour period, each data point is an average of 25 runs. Bold indicates best average time. The application and source_FO are running on UltraSparcs.

lem sizes. The window of n=10 performed best since the time window associated with 10 I/Os more accurately captured the Internet variance during the Complib runs in all but one instance (TG-12) [15]. This window outperformed both larger and smaller values of n. Clearly for this application, prefetching a single sequence (prefetch 1) provided the biggest performance boost (10-30%). However, the SFO was able to squeeze additional performance (~ 5%) out of the system by adapting to network and application characteristics.

4.3 MPEG

Currently, the most common way of viewing an MPEG file that is located on a remote file server is to download the entire file and then start the MPEG viewer. This solves the problem of unpredictable network access overhead for fetching MPEG frames, but introduces a potentially long latency into the viewing process. A better method is to overlap download and playtime as much as possible such as in streaming approaches. We developed a streaming SFO to interface between the MPEG player and a remote server to address this problem (Figure 11). The MPEG SFO is different from the other SFOs described thus far. In the matrix and gene sequence SFOs, the goal was to determine the optimal amount of prefetching between requests for data. With MPEG, the goal is to determine the smallest initial amount to prefetch such that the remainder of

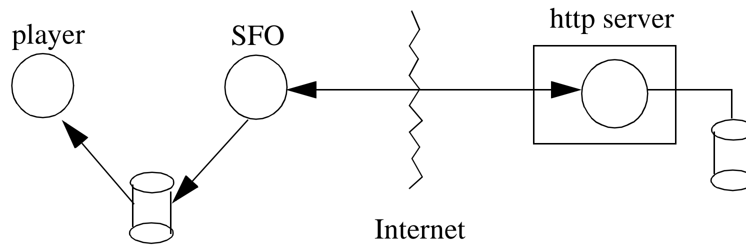

Figure 11: MPEG with SFO

the playtime can be fully overlapped with the retrieval of the rest of the MPEG file. Therefore, there is no need for an adaptive prefetching window. The MPEG SFO is written in C and uses TCP-IP communications. One of the goals of the MPEG SFO is to minimize the amount of code changes required to the MPEG player.

The MPEG SFO works as follows. When the user requests an MPEG from a web server, the SFO establishes a TCP-IP connection with the server, requests the MPEG file and stores the downloaded data in a local file. While the file is being downloaded, the SFO is measuring the rate at which it is receiving data and estimating the amount time the MPEG will play. Estimating playtime of an MPEG is complex. The SFO calculates what percentage of the MPEG must be downloaded before the player can begin and still be reasonably assured that the rest of the MPEG will be downloaded before it is needed. When that percentage has been reached, the SFO creates a second process. The parent process resets the handle to the beginning of the file before it returns it to the player. The player begins decoding and displaying the MPEG at this time. The child continues to download the file until it is complete. In essence, the file becomes a form of inter-process communication with one process reading from the head of the file while the other process is still writing to the end of the file.

The estimation method used by the SFO for T_{comm} is to take the total number of bytes downloaded and divide it by the time elapsed since downloading began. This method gives an accurate

picture of the actual download rate so far. Estimating the time an MPEG will play (T_{comp}) is far more complicated. The data collected showed most of the MPEG playtime falls into one of four steps: decoding, dithering, displaying or sleeping. The techniques used by the SFO to estimate the amount of time spent in each step are as follows: for each frame type, there is a linear relationship between the frame length and the number of CPU cycles it takes to decode them and a similar relationship between the decoded image size and the time required to dither and display the images. Linear regression was used to determine these cost equations for a given hardware platform. As the MPEG is downloaded, it is scanned for details such as frame type and length and image size. The details of these cost equations can be found in [3].

The MPEG SFO was tested on a suite of twelve MPEGs. The MPEGs came from different sources and, therefore, have different frame patterns, frame rates, resolutions, and picture size. The actual playtimes varied from ten seconds to over five minutes. On average, the SFO reduced total viewing time by 20% over downloading the entire MPEG file first. More importantly, an average of 50% of the download time is now overlapped with the MPEG playback. This means a much more pleasant viewing experience as the start latency is significantly reduced (Table 2).

The playback was also smooth in almost all cases. In only two of the sixty test runs, was the player was forced to pause for data. This can only be explained by a burst of traffic somewhere on the Internet between the two sites. Unlike traditional streaming approaches, no special protocols or server support was required.

5.0 Summary

This paper introduced the SFO paradigm for achieving optimized performance for remote file access. The SFO is based on the ELFS concept of files as typed objects to provide high-level interfaces and allow file specific properties to be exploited for performance. SFOs provide a prac-

	Total Download and Playtime			Time Before Playback Began		
Title	w/o Overlap	w/ Overlap	% Decrease	w/o Overlap	W/ Overlap	% Decrease
Blade	30.30	24.53	19.05%	12.16	6.38	47.49%
Blazer	363.81	335.29	7.84%	49.82	21.30	57.26%
Indy	49.00	34.62	29.35%	23.15	8.77	62.12%
lizard1	82.70	60.51	26.83%	35.18	12.99	63.08%
lizard2	57.74	44.23	23.41%	33.98	20.47	39.77%
Reichst	60.37	48.73	19.29%	20.17	8.52	57.75%
sci2	21.25	14.68	30.89%	11.31	4.75	58.04%
Ski	64.25	56.29	12.39%	13.68	5.71	58.22%
sts1	28.08	24.76	11.84%	7.68	4.36	43.26%
sts2	21.26	18.59	12.57%	9.91	7.24	26.97%
sts3	27.74	23.95	13.65%	9.60	5.82	39.43%
Ulysses	19.90	15.06	24.30%	8.85	4.01	54.67%
Average:			19.29%			50.67%

Table 2: MPEG results. All times shown are in seconds. The player was running on a Linux x86 machine and the remote server was running on an UltraSparc. The data is the average of five runs.

tical implementation of the ELFS ideas in dynamic wide-area networks. SFOs achieve their performance gain by exploiting features of the application and of the network. To date, we have applied our SFO prototype implementation to three small applications as a proof-of-concept: matrix operations, parallel gene sequence comparison, and a MPEG player each using a different server technology. The results indicate that the SFO paradigm can indeed improve application performance in a significant way. The SFO improved performance for the distributed matrix multiplication by 50% for reads and 20% for writes. For parallel gene sequence comparison, the SFO improved performance by 10-30%. Lastly for an MPEG player accessing movies across the Internet, the SFO reduced total playtime by 20% and reduced playtime latency by over 50%. In addition, insertion of SFOs into these existing applications required very minor code changes to the application.

6.0 Future Work

We recognize that a full-range of customized behaviors is not possible with the client-only architecture. Predictability and reliability will likely require server-side support. For example, server-side optimizations such as compressing or dropping data might be required for predictable performance. Similarly, supporting application-specific levels of reliability also requires server-side support since this is the logical place to replicate data or files. A *client-server* architecture in which SFOs have a client and server presence is being designed. The result will be a more integrated architecture for the SFO much like a true distributed file system with distinct client and server components. A key part of this architecture is the design of appropriate APIs for both the client- and server-side SFO. This API will allow an application to express its I/O needs to the SFO and allows application requirements to be communicated to the SFO which will enable the application to effectively customize the SFOs behavior. The server-side SFO will also store meta-data describing the layout of the remote file on disk and make this important information available via an API as well. The use of server-side SFOs also allows user-specified sharing semantics to be supported (e.g. session-semantics, Unix semantics, etc.). If several applications are accessing the same file then they would each have their own client-side SFO, but would share the same server-side SFO which would enforce the desired customized sharing semantics.

7.0 Bibliography

[1] T.E. Anderson et al., “Serverless Network File Systems, “*Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.

[2] I. Foster et al. “Remote I/O: Fast Access to Distant Storage,” *Proceedings of the Fifth Annual Workshop on I/O in Parallel and Distributed Systems*, 1997.

[3] M. Gingras and J.B. Weissman, “Smart Multimedia File Objects,” *1999 IEEE Workshop on Internet Applications*, July 1999.

[4] A.S. Grimshaw, “Easy to Use Object-Oriented Parallel Programming with Mentat,” *IEEE Computer*, May 1993.

[5] A.S. Grimshaw and W. A. Wulf, “The Legion Vision of a Worldwide Virtual Computer,”

Communications of the ACM, Vol. 40(1), 1997.

[6] A.S. Grimshaw, E.A. West, and W.R. Pearson, “No Pain and Gain! - Experiences with Mentat on Biological Application,” *Concurrency: Practice & Experience*, Vol. 5, issue 4, July, 1993.

[7] M. Harchol-Balter, A.B. Downey, “Exploiting Process Lifetime Distributions for Dynamic Load Balancing,” *SIGMETRICS*, 1996.

[8] J.K. Hollingsworth and P.J. Keleher, “Prediction and Adaptation in Active Harmony,” *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[9] J. Karpovich et al., “Extensible File Systems ELFS: Am Object-Oriented Approach to High Performance File I/O,” *Proceedings of the 9th OOPSLA*, 1994.

[10] T.M. Madhyastha and D. Reed, “Intelligent, Adaptive File System Policy Selection,” *Proc. of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, October 1996.

[11] R.L. Ribler et al., “Autopilot: Adaptive Control of Distributed Applications,” *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[12] M. Spasojevic and M. Satyanarayanan, “An Empirical Study of a Wide-Area Distributed File System,” *ACM Transactions on Computer Systems*, Vol. 14, No. 2, May 1996.

[13] A. Vahdat, “WebOS: Operating System Services for Wide Area Applications,” *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[14] G. Voelker et al., “Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System,” *SIGMETRICS*, 1998.

[15] J.B. Weissman, “Smart File Objects: A Remote File Access Paradigm,” *Sixth ACM Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[16] R. Wolski, “Forecasting Network Performance to Support Dynamic Scheduling,” *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997.

[17] R. Wolski, “Predicting the Availability of Time-shared Unix Systems,“ *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, 1999.