Smart File Objects: A Remote File Access Paradigm

Jon B. Weissman

Division of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
(jon@cs.utsa.edu)

Abstract

This paper describes a new scheme for remote file access called Smart File Objects (SFO). The SFO is an object-oriented application-specific file access paradigm designed to attack the bottleneck imposed by high latency, low bandwidth networks such as wide-area and wireless networks. The SFO uses application and network information to adaptively prefetch needed data in parallel with the execution of the application. The SFO can offer additional advantages such as non-blocking I/O, bulk I/O, improved file access APIs, and increased reliability. We describe the SFO concept, a prototype implementation in the Mentat system, and the results obtained with a distributed gene sequence application running across the Internet and vBNS. The results show the potential of the SFO approach to improve application performance.[1]

1.0 Introduction

Increasingly, applications require access to remote data sources including digital libraries, scientific databases, images, and movies, located on the Internet or across a wireless network. These applications range from Web browsers to data-intensive applications [1][3] to distributed supercomputing applications. Newly emerging *metacomputing* or *grid* environments promise to support many of these applications. These systems allow applications to access both geographically-separated computing and data resources with the goal of high performance [5][7][17]. We consider the problem of remote data access in which the data is structured in a set of one or more files. Access methods for remote files fall into one of several categories: download file to application location (staging), upload application to file location, or remote access. Staging has a number of drawbacks: it may require a large amount of disk space for certain kinds of files which may not be available, it takes time to download large remote files, the application may not need the entire file, and some files (e.g. satellite data) are constantly changing. Uploading (even if allowed) can be inefficient if the resources available at the file's location are inadequate to handle the application, and may not be appropriate for the application. For example, visualization of a remote data set requires that the data be shipped down to the application and not vice-versa. Remote access is a far more general and flexible method for file access. However, it is vulnerable to several potential problems due to the network environment: high network latency and low bandwidth, unpredictable network latency and bandwidth, and reliability.

We propose a new method for remote file access that addresses these problems in a flexible way. Rigid system-level solutions such as imposed by conventional distributed file systems are unlikely to be adequate for the wide-variety of applications and file types we expect to see. For example, multimedia applications may require predictable sustained performance for remote file access, while parallel scientific applications may require high performance for remote file access. We propose a new scheme for remote file access called Smart File Objects (SFO) that can accommodate the varying needs of applications. The SFO is user-level code or *middleware* that is placed between the application and the remote file, and acts on behalf of the application. It can mitigate the network performance problems by performing application-specific adaptive prefetching and overlapping, non-blocking I/O, and bulk I/O. The SFO can adapt to **both** the application and the network by exploiting information about each (a crucial capability in a dynamic network environment such as a *grid* or the Internet). It can also promote reliability and provide a more natural file access API for user applications.

This paper will introduce the SFO concept and its key features, show how it works, and describe a prototype implementation in the Mentat system [6]. We then apply the SFO paradigm to a distributed gene sequence comparison application that we have run across the Internet and vBNS. The results establish that real performance gains are possible using a SFO and serve as a proof-of-concept to show what is possible within the SFO paradigm.

1. This work was partially funded by grants NSF ASC-9625000 and CDA-9633299, AFOSR-F49620-96-1-0472, and ARP 010115-226.

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 [15]. xFS is a "serverless" file system designed to remove the bottleneck inherent in server architectures such as in Andrew [2]. 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 [16]. The SFO paradigm allows application-specific file access policies to be easily implemented in user-space better meeting an application's performance objectives. RIO is a file access paradigm for remote parallel file systems and shares our goal of high performance [4]. 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. The ELFS project is an object-oriented file system interface designed to attack the disk bottleneck for LAN-based parallel computing [11]. The SFO project has inherited some of the ELFS ideas, but is focused on attacking the network bottleneck in wide-area and wireless networks.

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 [14], and Active Harmony [10]. 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. The SFO employs adaptivity that is focused on the problem of remote I/O.

3.0 The Concept

The SFO is client code or *middleware* that interfaces between the application and the remote file. It runs on the "application-side" of the network and can be programmed to mitigate the poor network performance experienced in wide-area and wireless networks. We illustrate the SFO concept with read I/O (Figure 1). While the application is computing with prior I/O data, the SFO can 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. All components in Figure 1 are processes that communicate via RPC.

In the current implementation, the SFO has a single thread of control which means that it cannot both serve the application and perform remote I/O concurrently[2]. For this reason, the SFO must carefully "schedule itself" between prefetching activities and serving the application. The goal is to exploit opportunities for prefetching but not at the expense of delaying or blocking the application. A multithreaded SFO could dedicate a thread to serving the application and a thread to performing remote I/O. However, the use of threads add synchronization and scheduling overhead and still does not guarantee that the remote I/O thread will not delay the application serving thread. However, a multithreaded model will be examined in the future.

The SFO is programmed to be *application-* and *network-aware* in order to deliver the best possible performance for the application. 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 object, and the current application computation rate on each I/O data item by timing the interval between successive requests for data. With this information the SFO can adapt its prefetching and caching strategies. 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 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 shown above) 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

2. This is a property of the Mentat system.

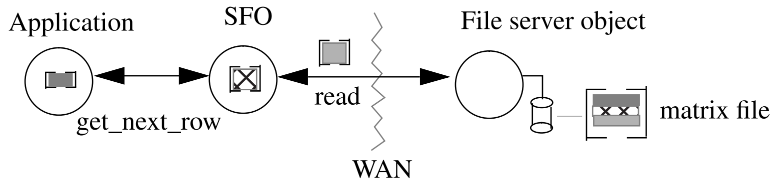


Figure 1: Matrix SFO in action

item should be similar and vary only if the application machine load varies. The predictions 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 and an exponential decay function is used to model this relationship. The weight associated with each prior value decays by a power of two. For example, with a time window of 5, the prior values would be weighted $\frac{16}{31}$, $\frac{8}{31}$, $\frac{4}{31}$, $\frac{2}{31}$, and $\frac{1}{31}$ (from most recent to oldest in time). In Section 4.1, we present experimental evidence to support this prediction model both for network bandwidth and application machine load (hence, application computation time). 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 object. 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 cost equations below are computed and used within the SFO. We expect the benefit provided by the SFO to be greatest when neither T_{comp} or T_{comm} dominate.

The "time window" for prediction is n, where n is the prior number of I/O operations. For the first few I/Os, $t < n$, so the window is shortened to t-1. The second term in (Eq.1) 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 upper-bound for i in Eq 1 is increased by the number of spikes encountered). The final equation gives the predicted number of data items to prefetch. Observe that it is adaptive to the computation and communication rates. We illustrate the performance of an adaptive SFO in Section 4.3. We point out that the machinery that we have just described could also be useful in a multithreaded SFO: predictions of T_{comp} and T_{comm} could be used to determine the scheduling duration of the prefetching thread.

The SFO can offer additional performance benefits: non-blocking I/O and bulk I/O operations. Non-blocking I/O increases the opportunity for overlapping between the SFO and the application. Non-blocking operations are easily supported in the SFO prototype implementation described next. Bulk I/O allows a SFO to collect a series of I/O requests perhaps from a parallel application and transform them into a single remote I/O request for efficiency as in RIO [4]. For example, suppose the application in Figure 1 was a parallel matrix computation composed of 10 processes each requesting a different row and the row size was 1000 bytes. Further suppose that the SFO was a parallel matrix SFO that knew about the structure of the application (e.g. that it was comprised on 10 processes). The SFO could then do a single bulk I/O request of 10 rows or 10000 bytes on behalf of all 10 processes rather than 10 individual remote I/Os. Writes could be handled similarly. The SFO could cache up writes and transmit them in an aggregate I/O operation. Given the high latency in a wide-area environment, the single larger I/O would likely be much more efficient. SFOs that support specific parallel file system interfaces such as MPI-I/O [17] could also be easily constructed. In summary, the SFO can offer improved performance in wide-area and wireless networks by:

- prefetching and caching to support overlap
- adapting to application and network performance
- providing asynchronous, non-blocking I/O
- providing bulk I/O

The SFO adopts an object-oriented interface that facilitates the construction of application-specific file classes. 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. 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 as we will show next.

3.1 SFO Prototype

We have implemented a SFO prototype in Mentat, an object-oriented parallel and distributed system [6]. Mentat was chosen for three reasons: 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 objects are important because they allow the SFO to operate asynchronously

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

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

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

```
persistent mentat class matrixSFO : basicSFO
{
    matrix_data *cache; // prefetched data
    int by_rows_or_cols; //controls prefetch
    ...
public:
    matrixSFO(int,int,int); //row,col,elt sz
    ~matrixSFO ();
    // read operations ...
    matrix_data *get_next_row ();
    matrix_data *get_next_block ();
    matrix_data *get_next_col ();
    set_interleave_factor (int);
    // write operations ...
    ...
};
```

```
persistent mentat class basicSFO
{
     file_object *remote_FO;
     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 2: MatrixSFO and BasicSFO interface.

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 are not germane to this paper, but can be found in [12].

The Matrix SFO of Figure 1 can be easily implemented in Mentat (Figure 2). The `matrixSFO` class derives from a base SFO, called `basicSFO`, that provides basic file operations such as `open`, `close`, `get_file_size`, etc. The `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. Notice that the `matrixSFO` provides a high-level interface that more naturally reflects the way in which the user might wish to access the file, e.g., `get_next_row`, much like in ELFS. Implementation fragments for the SFO and for a client application are shown in Figure 3. Non-blocking I/O is supported in the application by delaying use of the return result until needed (Figure 3b).

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 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, and then finally uses the row in a dot product. This example is meant to be illustrative of how an SFO can be defined, implemented, and used, and has omitted some details for brevity. We believe that it will be possible to develop SFOs for common classes of files using inheritance that can offer desirable APIs and performance characteristics. We focus on the performance properties of the SFO in the remainder of this paper. We performed an experimental evaluation of a distributed gene sequence comparison application called Complib in two wide-area networks: the Internet and the vBNS as a test case.

4.0 Results

4.1 Network Environment

We performed a series of network experiments to study Internet and vBNS communication performance as a basis for programming the SFO described in the next section. The network environment consists of three sites, University of Texas at San Antonio (UTSA), University of Virginia (UVa), and the University of California, San Diego (UCSD). We used two configurations: UTSA and UVa (Internet), UVa and UCSD (vBNS). The vBNS results represent a one-day sampling of the average network bandwidth via *ping* for a 4K message[4] (Figure 4a). These results indicate that the vBNS bandwidth is fairly consistent and predictable with an average communication time of 86 ms and a standard deviation of 3.5 ms. The Internet results represent a two-day sampling of the average network bandwidth via *ping* also for a 4K message

3. Mentat together with extensions for wide-area computing such as the Legion file system were the prototype for the new Legion system [7].

4. This is the message size for the Complib application.

```
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 ...
   }
```

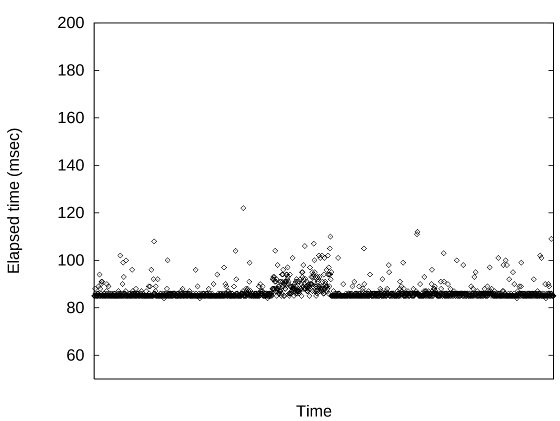
(a) MatrixSFO implementation

```
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];
        ...
}
```

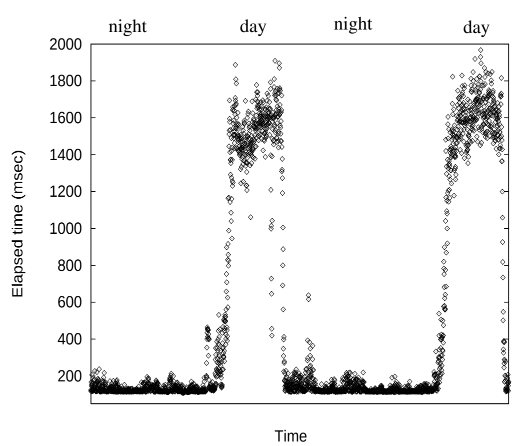
(b) application fragment

Figure 3: MatrixSFO and application fragment

(Figure 4b). These results show the wide-variance in Internet performance from the low-bandwidth high-congestion regime during the day, to the higher-bandwidth low-congestion period overnight. During the day the average comminication time is 1263 ms (499.6 ms standard deviation), and during the night the average time is 134 ms (44.0 ms standard deviation) for the 4K message. The high standard deviation suggests that accurate prediction of Internet communication time may not be possible. However, there is a predictability to the Internet's performance along several time-scales. Long-range prediction of average bandwidth is possible ignoring occasional congestion spikes: the daytime and evening numbers are generally predictable from day to day. This daytime vs. evening prediction is on a time-scale that may not be useful unless the application was long-running (hours to days) and accessing remote data throughout its execution. However, we have determined that an accurate prediction may be possible on shorter time-scales. We have observed that the Internet bandwidth values are similar along time-scales of seconds to minutes in the **overnight regime** (Figure 5). We term these regions *clusters* and define a cluster to be a time window in which the variability among observed bandwidth values is small (e.g. the points in cluster 1 fall between 100 and 130 ms, a 30% difference). This is a conceptual definition in that the SFO does not detect clusters per se, but the existence of clusters suggest that accurate prediction may be possi-



(a) vBNS



(b) Internet

Figure 4: Communication performance for vBNS and the Internet. UTSA has a relatively poor Internet connection.

ble. The values within a cluster appear to be similar and predictable, but different clusters can exhibit a variability of ~2 times (from an average of ~110 ms in cluster 1 to ~220 ms in cluster 8 for the 4K message). Given this data, we have chosen a value of k=3 be used for spike removal in (Eq. 1) since a factor of 3 appears to exceed the typical observed overnight variance range (Figure 5). This data also suggests an important opportunity: when the bandwidth rises, a SFO can detect this and prefetch more aggressively knowing that the measured bandwidth will be sustained for a short period into the future, and when the bandwidth falls, prior cached data can be used avoiding expensive network access. This is captured by Eq 1. In this situation, more data can be prefetched during cluster 1 (at ~110 ms), avoiding more expensive access during cluster 2 (at ~220 ms). The clusters most likely reflect intermittent periods of congestion or congestion relief. In contrast, the vBNS data did not exhibit significant clustering most likely due to the lack of congestion in the vBNS. Exploiting bandwidth variability

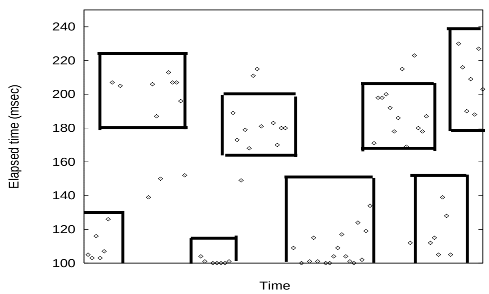


Figure 5: Communication performance for the Internet during an overnight interval (5 minute sample). Clusters are groups of similar values shown as boxed points. Eight clusters can be

is possible for the Internet but probably not for vBNS until it becomes more heavily used. However, a second opportunity exists independent of the network: machine load variability. When the load on the application machine rises, there is a greater opportunity for prefetching since the application will spend more wall-clock time between requests for remote data. We believe that recent machine load is predictive of future machine load much like Internet bandwidth along time-scales of interest to us. Much of the research in dynamic load balancing in shared networks is based on this reasonable assumption [9]. These prediction relationships are captured in (Eqs. 1-2). The potential variability and predictability of network bandwidth and application machine load creates a window of opportunity to further improve the performance of a SFO: prefetching can be made adaptive to network and application characteristics. We present data to substantiate this claim for a real distributed application, Complib, next.

4.2 Complib

Biological sequence comparison is concerned with the classification of genes that have been determined by DNA cloning and sequencing techniques. Because it is difficult to determine the function of a given protein, a newly sequenced gene is compared with other genes that have evolved from a common ancestor. If the two genes are similar then it is likely that they perform similar functions [4]. DNA is composed of four nucleotide bases (A, C, G, T) which encode the 20 amino acids that comprise proteins. Comparing protein or DNA sequences is a string matching problem over strings of base pairs. Through the Human Genome Initiative, DNA and protein libraries are available for most published sequences. The comparison problem is a computationally intensive process.

Complib is a parallel/distributed Mentat application that compares a source library of sequences to a target library of sequences. The libraries are ascii text files. Complib utilizes several heuristics for string matching, but we have chosen to use Fasta, a fast heuristic that sacrifices some accuracy for speed. The details of this algorithm may be found in [4]. 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. This process repeats for each source sequence. The workers are arranged in a tree with the leaves performing the computation (Figure 6). The results contain a score for the current source sequence generated by each worker based on a comparison to its target sequences. In a metacomputing environment, it is possible 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 7). The source library is located remotely via the `/legion` file

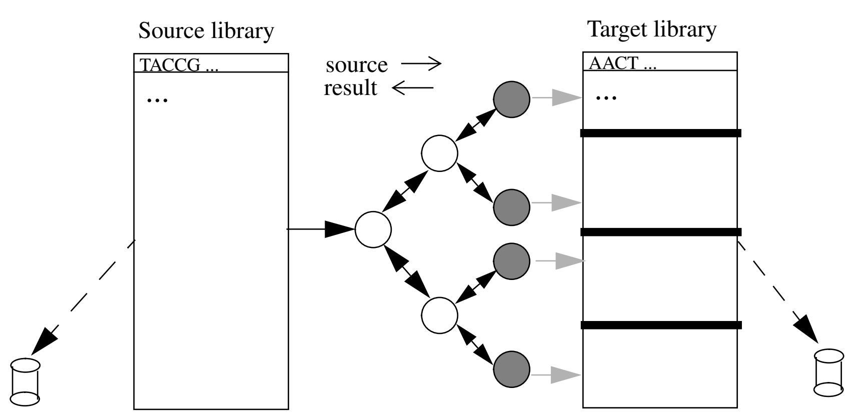


Figure 6: Complib structure

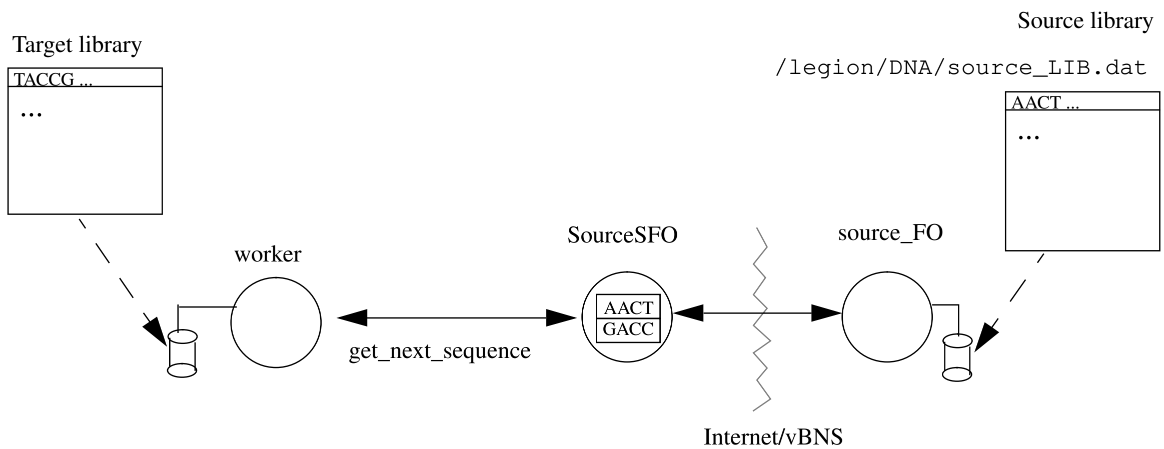


Figure 7: Complib with SFO. The application, SourceSFO, and target library are all located on the "fast" side of the network. The source library file is located remotely in `/legion` and managed by source_FO. Access to the source library file is controlled by the SourceSFO which adapts its behavior to the network and application. The application consists of a single computation worker.

system with access controlled by a SFO, `SourceSFO` that derives from `basicSFO`, and we used a single worker for simplicity. The basic 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. Parallel workers would be no problem since the `SourceSFO` would simply perceive a faster computation rate. The target library and worker 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 worker. 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 following four modes: (1) download first, (2) remote access, (3) prefetch 1 sequence, (4) adaptive prefetch. The download first option was implemented by the developers of Complib – it first downloads the entire 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.

4.3 Complib Performance

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. We used a single Sparc workstation in each site, a Sparc-5 at UCSD, a SparcServer 20/514 at UVa, and an UltraSparc at UTSA. As the vBNS did not exhibit significant bandwidth variation, there is little gain to be expected by adapting to changing bandwidth. Instead, we used the vBNS testbed to experiment with adaptation to changes in application machine load. We put background load on the UVa machine to determine whether the SFO adapted to changes in the computation rate, and whether any performance gains could be realized. The background processes alternated between computation and sleeping to create a load variance. On the Internet, we ran overnight on unloaded machines during

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: Internet 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.

Problem instance	Download first	Remote Access	Prefetch 1	Adaptive Prefetch (n=5)	Adaptive Prefetch (n=10)
TG-12	19121	14339	13867	**13157**	13256
TG-25	25260	17206	16605	**15676**	15721
TG-50	32319	26928	23453	23509	**23364**
TG-100	38301	36912	**29177**	29468	29224

Table 2: vBNS 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.

periods of low congestion as shown in Figure 5 since there is not sufficient bandwidth during the day. The results are summarized in Tables 1 and 2 for Internet and vBNS respectively.

The execution times for the Internet testbed are slightly better than on the vBNS testbed due to a faster application computer (UltraSparc vs. SparcServer 20/514). The results in both networks exhibit similar trends. 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 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. We ran the `SourceSFO` with a "time window" of n=5 and n=10 to see if any further gains were possible. The results indicate that performance could be further improved for appropriate problem sizes. On the Internet, we see that the performance gain due to adaptation varied from 400 ms to over 1 sec depending on the problem. A window that contained 10 I/O operations yielded better performance than the smaller window. We speculate that this reflects the clustering in Figure 5 − the cluster time windows appear to contain closer to 10 I/O operations. We also believe that greater performance benefits will be seen for longer running applications due to an increasing number of clusters to exploit. Future research is needed to study the window size for network adaptation.

On the vBNS, we ran background processes to load the application machine during ALL runs. While the background processes were computing and slowing down the sequence comparison, the SFO was able to automatically prefetch more data due to increased overlap opportunities. Later when the background processes were sleeping the wall-clock speed of the worker increased due to a larger share of the CPU, but the SFO was able to deliver cached data and "keep up" with the application demand. This phenomena is best seen for the smaller problems, TG-12 and TG-25 in which a gain of 700-900 ms is seen. For the larger problems, computation already dominated communication and there was little room for adaptation due to load variance. The imposition of load on the machine did not create any additional opportunities for prefetching since the SFO was able to prefetch the next sequence without delaying the application even when the machine was unloaded. For these experiments, the choice of window size appeared to be less important than on the Internet.

Clearly for this application, prefetching a single sequence (prefetch 1) provided the biggest performance boost. However, the SFO was able to squeeze additional performance out of system by adapting to network and application characteristics. The results illustrate the benefits of the SFO approach for this application and serve as a proof-of-concept for the SFO paradigm in general.

5.0 Summary and Future Work

Efficient, predictable, and reliable access to remote files has become increasingly important. A solution to this problem must be flexible enough to accommodate the wide range of file types and applications that are beginning to emerge in wide-area and wireless network environments. The SFO achieves this flexibility by adopting an object-oriented middleware solution that is easily tailored to the application and file characteristics. SFOs exploit dynamic application and network information, and are able to perform adaptive prefetching and caching, bulk I/O, and non-blocking I/O, to mitigate the problems of low bandwidth, high latency, and unpredictable network performance. The performance potential of the SFO concept was demonstrated by our prototype implementation and application to Complib. While we believe that the SFO concept can yield general file classes that can be useful to applications, future research is needed to demonstrate this potential.

We plan on developing parallel file classes with support for bulk I/O to improve performance for parallel applications. Another future area is reliability. We believe reliability can be improved in an application-specific way using the object-oriented paradigm. For example, a SFO could create multiple remote file object replicas to improve availability. We are currently working on a multimedia SFO that sits between an mpeg viewer and remote image files to improve performance in wide-area, wireless, and ISDN networks. Future work also includes refining and improving the basic SFO infrastructure needed by all SFOs. Along these lines, we are planning to incorporate more sophisticated prediction strategies for computation and communication performance to improve the adaptive behavior of the SFO as in [18]. Finally, we plan on investigating the benefits of multithreading within a SFO.

6.0 Acknowledgments

Thanks to Bob Hiromoto for a careful reading of this manuscript and helpful suggestions. This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. ARP 010115-226. The author also wishes to acknowledge the Computer Science departments at the University of Virginia and the University of California, San Diego, for making remote resources available to our research.

7.0 References

[1] A. Amoroso, “Wide-area Nile: A Case Study of a Wide-area Data-parallel Application,” *The 18th International Conference on Distributed Computing Systems*, 1998.

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

[3] DOCT. URL: *http://www.sdsc.edu*, 1998.

[4] 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.

[5] I. Foster and C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit,” *International Journal of Supercomputing Applications*, 11(2), 1997.

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

[7] A.S. Grimshaw and W. A. Wulf, “The Legion Vision of a Worldwide Virtual Computer,” *Communications of the ACM*, Vol. 40(1), 1997.

[8] 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.

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

[10] 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.

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

[12] Legion team, “Legion 0.1 User’s Manual”, University of Virginia, 1996.

[13] MPI-IO. URL: *http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html*, 1998.

[14] 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.

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

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

[17] J.B. Weissman, “Gallop: The Benefits of Wide-Area Computing for Parallel Processing,” *Journal of Parallel and Distributed Computing*, 54(2), November 1998.

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