Armada: A parallel file system for computational grids

Ron Oldfield and David Kotz

Department of Computer Science
Dartmouth College
Hanover, NH 03755
{raoldfi,dfk}@cs.dartmouth.edu

Abstract

High-performance distributed computing appears to be shifting away from tightly-connected supercomputers to "computational grids" composed of heterogeneous systems of networks, computers, storage devices, and various other devices that collectively act as a single geographically distributed "virtual" computer. One of the great challenges for this environment is providing efficient parallel data access to remote distributed datasets. In this paper, we discuss some of the issues associated with parallel I/O and computatational grids and describe the design of a flexible parallel file system that allows the application to control the behavior and functionality of virtually all aspects of the file system.

1 Introduction

An exciting trend in high-performance distributed computing is the development of widely-distributed networks of heterogeneous systems and devices, known as *computational grids*. Applications for these environments use high-speed networks to logically assemble collections of resources such as scientific instruments, supercomputers, databases, and so forth. One important challenge facing grid computing is efficient parallel I/O for grid applications. This effort is particularly challenging because most earlier research for parallel I/O and parallel file systems targeted tightly-connected computer systems that are much different than grid environments.

Traditional parallel file systems for tightly-connected computer systems typically restrict control of the functionality to the kernel and privileged servers. Applications not trusted by the operating system are limited to the interface and implementation provided by the privileged software. Computational grids, unlike tightly-connected systems, have heterogeneous components, unpredictable performance variations, and span multiple administrative domains. In such an environment, centralized control of resources and functionality is not practical or efficient.

This work was supported by Sandia National Laboratory under contract DOE-AV6184.

The Galley parallel file system [14] proposed that the traditional functionality of parallel file systems be separated into two components: a fixed core, standard on all platforms, that encapsulates only primitive abstractions and interfaces, and a set of high-level libraries to provide a variety of abstractions and application-programmer interfaces (APIs). While this approach is more flexible than conventional parallel file systems, the core API still resides on the compute nodes. Applications have no control over the policies implemented on the I/O nodes. The Armada parallel file system adds this flexibility by allowing application-supplied code to run on the compute nodes, I/O nodes, and intermediate network nodes. The system decides where to place the code based on resource availability, cost, performance, or user preference. We illustrate the evolution from conventional parallel file systems to Galley and then to Armada in Figure 1.

In Armada, the core file system is extremely simple: there is no caching, prefetching, or remote access. The data servers provide a (local) interface to open, close, read and write data and arbitrate among server programs competing for processor time, memory, disk access, and network access. In short, the core system focuses on the shared aspects of the file system.

There are many reasons to allow application code to run outside the client node. I/O nodes can implement application-specific caching and prefetching policies to improve disk usage. Data-dependent mapping functions can distribute file data in an application-specific way, for example, in applications with a data-dependent decomposition of unstructured data [6]. I/O nodes can apply data-reduction filters to save network bandwidth and compute-node mem-

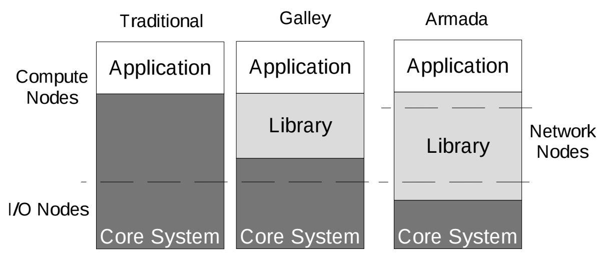


Figure 1. Our proposed evolution of parallel file-system structure allows application-control over the functionality on the compute nodes, the I/O nodes, and intermediate network nodes.

ory [3]. I/O nodes can rearrange file blocks between disks without passing data through the client nodes. In short, there are many ways to optimize memory and disk activity on the data servers, and reduce disk and network traffic, by moving what is essentially application code closer to the data.

2 Armada system design

We designed the Armada parallel file system for grid applications that need access to large, remote, distributed datasets. Users access remote datasets through a network of distributed application objects. Each object, known as a *ship*, provides a small piece of functionality that helps define the behavior and structure of the overall system. Applications can move functionality closer to the data by executing ships on remote hosts. The goals of our design are to provide a simple architecture for building and maintaining the application-specific ships, and to provide a secure, robust environment on which to deploy the ships.

2.1 Architecture

In Armada, the core functionality of the file system lies on the processors that host ships. The system code, called a *harbor*, is layered on top of the host's operating system and can exist on machines near the client (client nodes), or on machines near the data (I/O nodes), or on any machine between the clients and the data (network nodes). In all cases, harbors provide a secure execution environment that allows ships to access the host CPU, memory, and network resources. Harbors on an I/O node provide additional functionality to access to the host file system.

2.2 Security

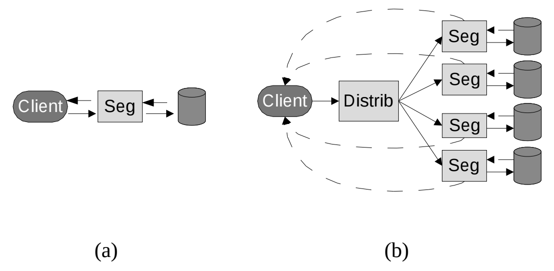
Armada provides authentication and authorization services through a security manager known as the *harbor master*. Before installing an untrusted ship on a harbor, the harbor master authenticates the client wishing to install the ship and authorizes use of the host resources based on the identity of the client and on the security policies set by the host. We discuss the details of our implementation in Section 4.1.

2.3 Blueprints

The metadata used to describe the arrangement of ships on the network is called a *blueprint*. Unlike traditional file systems, Armada does not provide specific storage or a namespace for this metadata. Blueprints can be stored on conventional file systems, web servers, databases, or any other location; one typical approach would be to store blueprints in files on a Unix workstation. The system storing the blueprint provides the necessary access-control mechanisms for the blueprints, and Armada provides security at the harbors to prevent unauthorized clients from accessing data represented by the blueprints. We do not prevent authorized clients from accessing data through arbitrary blueprints; however, the application will likely need to know the correct organization and layout of the data to use it effectively. This approach is not unlike the approach used by most file systems. For example, a Unix application may access data within a Unix file as sequence of bytes, even though the format of the data may be more structured. Users and applications typically infer the format of a Unix file through a filename suffix (e.g., .gif, .tar, .zip), a file header, or through prior knowledge by the application.

Figure 2 shows examples of some simple blueprints. In the blueprint illustrations in this document, we show the clients on the left and the servers on the right, with Armada ships between. The first blueprint provides access to data stored on a single disk. The second blueprint represents a distributed file, and the third blueprint shows how an application might modify the second blueprint to include caching functionality and an application-specific interface.

In the examples in Figure 2, solid lines show the control flow (the flow of requests), and the dashed lines show the data flow for reading the file. In a proper implementation, each ship should forward requests, split and forward requests, or generate new requests. For example, the cache ship from Figure 2-c generates a new data request every time a cache miss occurs. The distribution ship splits requests from the cache into smaller sub-requests and forwards the sub-requests to the appropriate segment ship (we discuss segment ships in Section 3.1). While the control flow includes every ship in the graph, data only needs to pass through ships that generate new requests. For exam-



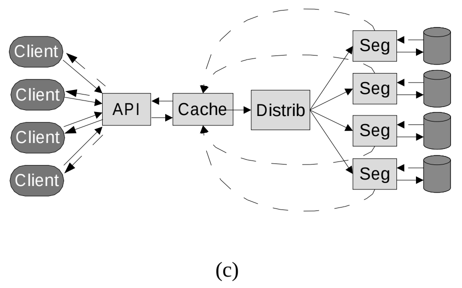


Figure 2. Example blueprints in Armada: (a) shows a blueprint providing simple access to a single file segment; (b) shows a blueprint providing access to a distributed file; (c) extends the blueprint from (b) by adding an application-specific cache and interface.

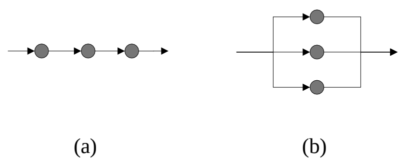


Figure 3. The types of subgraphs in a series-parallel digraph: (a) shows a sequential subgraph and (b) shows a parallel subgraph.

ple, the data does not need to flow through the distribution ships from Figure 2-c, but it should pass through the cache and the API ships. An implementation that allows data to bypass some of the ships will make fewer data copies and remove potential bottlenecks in the network.

We can express any legal control-flow graph as a series-parallel directed tree (SP tree). In an SP tree, a vertex is either a ship or a subgraph. Subgraphs take two forms: a sequential subgraph, or a parallel subgraph. A sequential subgraph (shown in Figure 3-a) represents a series of connected vertices. A parallel subgraph (Figure 3-b) consists of a set of simultaneous vertices, connected to a source on the left and a sink on the right. By expressing the arrangement of ships in series-parallel form, we convert a potentially complex network graph into an easier to manage tree. Figure 4 illustrates this conversion.

The data-flow graph is technically a subgraph of the control-flow graph and can be expressed as a separate series-parallel tree. Rather than supporting two separate graph representations, it is often more convenient to establish the data-flow graph at runtime. In Section 4.3, we discuss such an implementation.

2.4 Accessing data in Armada

Unlike traditional file systems that allow access to data with a single "open" call, Armada requires a two-step process. First, the programmer constructs a blueprint describing the arrangement of ships in the graph. Second, the system deploys the ships to the network and thus provides access to the remote data. We discuss each of these steps in turn.

There are two ways to construct ship graphs in the Armada system: the programmer can build them at runtime using a graph-construction library, or the user can build them off-line using a graphical interface. Both the runtime library and the graphical interface allow the application, or user, to construct and connect the various types of SP-tree vertices. One can construct a blueprint from scratch, or download an existing blueprint and dynamically add application-specific functionality by attaching new ships to the existing graph. For example, an application may want to apply a filter near the data, or access data through a different interface. In such a case, the application could construct a new two-vertex series object, with the application-constructed graph as the left vertex and the downloaded graph as the right vertex.

After constructing the blueprint, the application tells the system to deploy the ships to the harbors. Our algorithm recursively traverses the SP tree, starting at the root, and checks for three cases: a ship, a series, or a parallel vertex. If the vertex is a ship (the base case), the algorithm finds the harbor specified by the ship representation, installs the ship on the harbor, establishes communication links between the new ship and the list of ships to the right, and returns the ship as a list of size one. If the vertex is a series, we recursively deploy and connect the vertices in the series from right to left and return the list of ships from the left-most vertex. In the third case, we recursively deploy the vertices in the parallel group and return the cumulative list of ships deployed by those vertices.

3 Application-specific functionality

The task of the application or library developer is to decide what functionality is required and to implement that

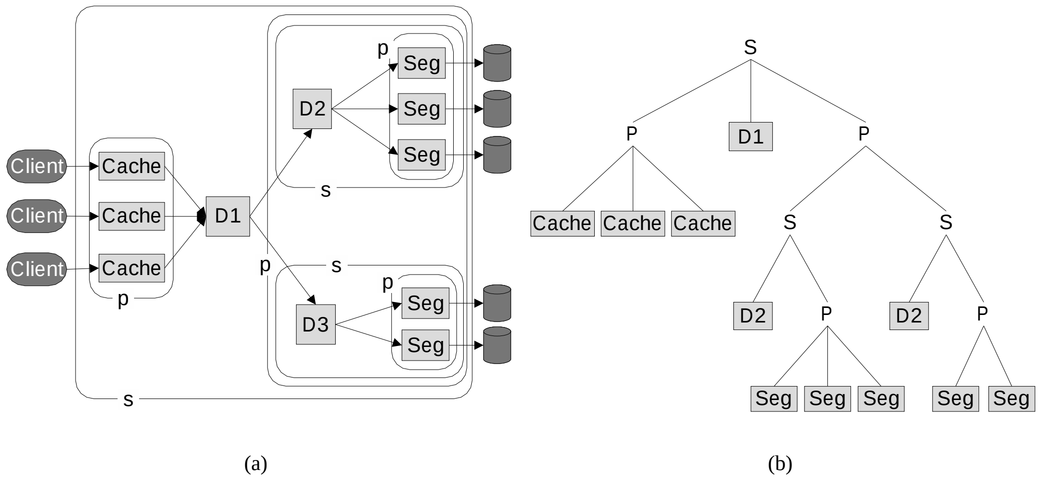


Figure 4. Subfigure (a) shows an Armada blueprint broken into series-parallel vertices. Subfigure (b) shows the series-parallel tree representation of the Armada blueprint.

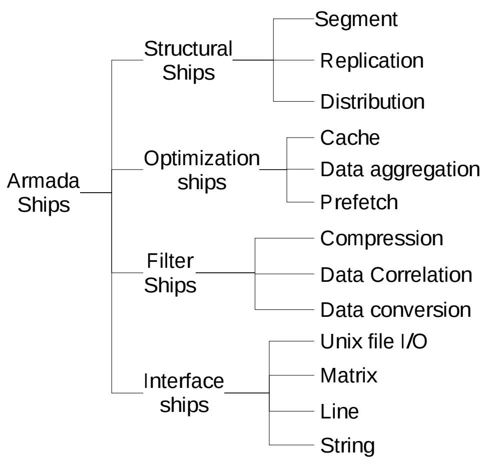


Figure 5. Hierarchy of Armada ship classes.

functionality in the form of Armada ships. To aid the developer, the class hierarchy in Figure 5 illustrates some basic functionality.

In our hierarchy, Armada ships fall into one of four categories: structural ships that describe the organization and layout of the data, optimization ships that attempt to improve performance, filter ships that manipulate the data, and interface ships that provide semantic meaning to the data. We discuss each type in turn.

3.1 Structural ships

Structural ships describe the organization of data onto distributed data servers. They can describe application-specific distributions of parallel files, provide replication management, or combine separate existing datasets into federations of datasets. In our proposed ship hierarchy, we present three types of structural ships: *segment* ships, *distribution* ships, and *replication* ships.

Segment ships interact directly with remote storage servers to provide access to file *segments* stored on the data servers. A segment is a sequence of bytes, typically stored as a file in a local conventional file system. Typically, the segment ship object is loaded on the harbor containing the segment data. Implementations of segment ships process requests by reading and writing data through the local file system interface provided by the harbor.

Distribution ships define an application-specific mapping of data to other ships. A specific example would be a distribution ship that provides a simple striping algorithm for mapping byte requests to an array of segment ships. Another implementation could use a table, instead of a striping algorithm, to map portions of an incoming request to lower-level ships. Developers could also implement complex "nested" distribution strategies by layering distribution ships on top on one another. For example, consider an application that sums the contents of two existing distributed files and sends the result to the client. We illustrate the graph for such an application in Figure 6. Our application uses two types of distribution ships: a distribution ship to direct requests to the proper file (labeled D1 in the figure), and a distribution ship to direct requests to segment ships to retrieve the portions of the file needed by the application (D2 in the figure).

A third type of structural ship provides support for replication of data. Replication can improve performance and

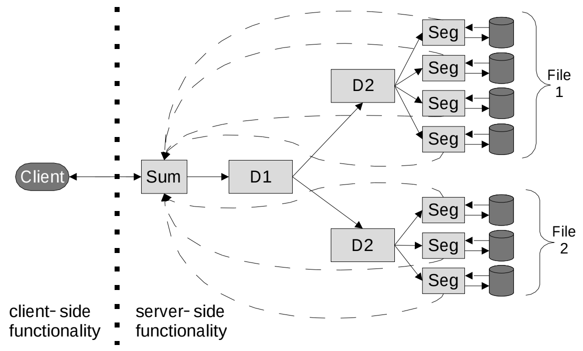


Figure 6. The graph shows application that uses nested distribution ships to sum the contents of two separate distributed files.

fault tolerance for "read-mostly" datasets. By increasing the number of locations of the data set, we reduce the chance of the data being unavailable because of a network or server failure. We can also use replicas to distribute load by forwarding requests to the least-used storage server, or to the server with the fastest network connection. Such an implementation could use performance tools like the Network Weather Service [18] or Netlogger [16] to decide where to forward requests.

3.2 Optimization ships

Optimization ships improve I/O performance by reducing data retrieval time through traditional bandwidth-reduction and latency-reduction techniques. Ships of this type provide caching, data aggregation, and pre-fetching. Caching and pre-fetching policies that match the access patterns of the application can significantly reduce network latency associated with reading a remote file. Data aggregation, near the client nodes for writing and near the I/O nodes for reading, increases the size of network transfers and thus reduces the number of requests that travel through the network.

3.3 Filter ships

Filter ships provide data-processing functionality between the client and the data servers. Data-reduction filters applied near the data source could improve performance by reducing the amount data sent through the network. For example, consider again the application in Figure 6. In that application, a filtering ship sums the contents of two existing files and sends the result to the client. By applying the filter near the data sources, we reduce the amount of data transferred through the network by half.

3.4 Interface ships

With the goal of matching the application semantics, Interface ships provide a high-level interface to an Armada file. It is often the case that well-structured typed objects have predictable access patterns and by representing a file as a high-level object, rather than a traditional "flat" sequence of bytes, optimizations like pre-fetching and caching can be effectively built into the file structure [5]. Parallel applications could benefit from a parallel I/O interface that allows optimizations such as collective I/O and data sieving [15]. Others may provide an interface customized for a language designed for out-of-core programming [2]. Still others may want to provide a conventional Unix I/O interface to allow legacy software to seamlessly access Armada files.

4 Implementation

Our design allows library programmers to define the policies and access interfaces for the system. Armada must provide the core functionality to support the libraries. In particular, we need a secure environment for executing untrusted client ships on remote harbors, we need mechanisms to construct and append blueprints, and we need to be able to deploy and connect application ships.

4.1 Security

Security on Armada harbors requires authentication of clients, protection of the system resources from untrusted ships, and control over access to resources allocated to ships.

The harbor master uses authentication mechanisms, provided by the host machine, to identify clients that wish to install ships on the harbor. The host provides mechanisms that implement security policies set by the host administrator. Although this issue is beyond the scope of this paper, two options for implementing authentication include using ssh, or using a Kerberos authentication service [13].

The most common approaches used to protect system resources from untrusted code are hardware protection (e.g., running the untrusted code in a separate Unix process), software fault isolation (SFI) [17], verification of assembly code [10, 11], and use of a type-safe language (e.g., Java or Modula3 [12]). Hardware protection requires untrusted code to run in a separate address space from the harbor. While this clearly protects the harbor from the client code, the overhead of communicating through normal IPC system calls is quite high. Both SFI and verification of assembly code offer promising solutions, but they typically target a limited set of machines, making them non-portable. Type-safe languages provide portability and memory protection for untrusted code: two important features for heterogeneous grid environments.

We chose to use Java because it provides a "sandbox" for executing untrusted client code on the harbors, it is reasonably efficient now that just-in-time compilers are available, it is increasingly popular among HPC programmers, it has convenient mechanisms for remote execution and communication (RMI), and it can interface to application code in other languages through the Java Native Interface (JNI). Only the ships and harbors need to be written in Java; client code could be in C, C++, or possibly others.

Even though Java provides protection against unauthorized memory accesses, it still does not provide the resource-control mechanisms we desire. In particular, Java does not allow object references to be revoked. A ship implemented in Java could allocate a resource and hold on to it much longer than it needs to, thus denying access to that resource from other ships. We remedy this by forcing ships to access resources through harbor-generated capabilities [4]. Capabilities provide a "wrapper" around system resources that allows harbors to revoke access to client ships. When a ship accesses a revoked resource, the capability throws an exception that is then handled by the ship implementation.

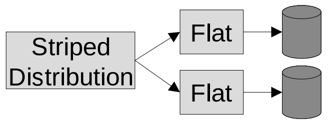
4.2 XML blueprints

As we explained in Section 2.4, the first step towards gaining access to Armada files is constructing a blueprint. We implement blueprints as Extensible Markup Language (XML) documents. We chose XML primarily because the Java API for XML Parsing (JAXP) provides existing mechanisms for creating and manipulating XML documents, and because XML's hierarchical structure fits our SP tree needs well.

As specified in Section 2.3, blueprints use series-parallel trees to represent ship graphs. A vertex in the SP tree is either a representation of a ship, a series of connected vertices, or a set of parallel vertices. We define the structure of the SP-tree blueprint with this XML Document Type Definition (DTD):

```
<!ELEMENT blueprint (vertex | ship)>
<!ELEMENT vertex (vertex | ship)+>
<!ATTLIST vertex
      type  (PARALLEL,SERIAL) #RE-
QUIRED >
<!ELEMENT ship EMPTY>
<!ATTLIST ship
      host      CDATA #REQUIRED
      className CDATA #REQUIRED
      fileName  CDATA >
```

A ship representation in a blueprint contains information about the preferred host and information about how to locate and construct the ship on the chosen harbor. Figure 7 shows an XML blueprint for a simple distributed file.



```
<blueprint>
   <vertex  type="SERIES">
      <ship
         host="tahoe.cs.dartmouth.edu"
         className="StripedDistribution" >
      </ship>

      <vertex type="PARALLEL">
         <ship
            host="dorset.cs.dartmouth.edu"
            className="Flat"
            fileName="test1" >
         </ship>

         <ship
            host="pecos.cs.dartmouth.edu"
            className="Flat"
            fileName="test1" >
         </ship>
      </vertex>
   </vertex>
</blueprint>
```

Figure 7. XML blueprint of a simple distributed file.

4.3 Deploying ships to harbors

We implement harbors and ships as remote objects that communicate using Java Remote Method Invocation (RMI). Harbors are persistent server objects that export methods to "anchor" Armada ships to the harbor. The anchor methods require a class name and an optional class loader.

```
public interface Harbor extends Remote {
    Ship anchor(String className)
      throws RemoteException;
    Ship anchor(String className,
                ClassLoader loader)
      throws RemoteException;
}
```

We deploy ships to harbors by executing a Java version of the deployment algorithm from Section 2.4 on a client thread. In our implementation, the client installs a new ship on a remote harbor by calling the harbor's "anchor" method. The harbor returns a reference to the ship so that other ships can communicate using RMI.

Ships pass three types of objects between each other: requests, data, and exceptions. Request objects travel from the client to the server through ships along the control path. They describe the type of data and contain information about the size and location of requested data. They also hold a reference to the ship or client that generated the request. For example, a byte request would have an offset, a length, and a reference to the ship that constructed the request.

As we mentioned in Section 2.3, each ship should forward requests, both split and forward requests, or generate new requests. A ship can "split" a request by duplicating the request and modifying the access information of the duplicates. In our byte-request example, duplicates would have the same reference to the ship that generated the original request, but different values for the offset and length.

We establish a data-flow path between ships and the data servers by using the ship reference in the request object. The requesting ship implements an interface to send and receive data accessed by the segment ships on or near the data servers. By using this approach, we bypass ships in the control path that do not explicitly generate new requests. Our approach is similar to the disk-directed I/O model [7] because the segment ships, located on the servers, initiate the data flow. Data is either "pulled" to the server by the segment ships, or pushed toward the client by the segment ships.

5 Related work

Various groups within the research community as well as the commercial sector are investigating issues related to I/O for computational grids. This section discusses the projects, systems, and ideas that had the largest influence over the design of the Armada system. In particular, we discuss parallel I/O systems and systems with support for remote user code.

The Hurricane File System (HFS) [8] is a parallel file system for tightly-connected shared-memory parallel machines. It was designed with the notion that flexibility is essential to provide good performance to a wide range of applications. They provide this flexibility by representing files as a collection of application-specific stackable building blocks. They demonstrate that for a several file access patterns, HFS can provide the full I/O bandwidth of the disks to the application. Armada uses the same philosophy of flexibility, but we target a grid environment.

An interesting system that uses a data-flow model is the Parallel Storage-and-Processing Server (PS^2) [9], from École Polytechnique Fédérale de Lausanne. PS^2 uses the Computer-Aided Parallelization tool (CAP) to express the parallel behavior of the I/O intensive applications at a high-level. The CAP system constructs a data-flow computation graph with "actors" as nodes of the graph. The actors are computational units that provide application-specific functionality. For I/O-intensive applications, the actors provide application-specific data distribution, prefetching, or filtering that potentially execute near the data storage devices. PS^2 presents a nice programming model for data-intensive applications; however, they lack the flexibility to provide application-specific interfaces and the user has no control over the placement decisions of the system.

DataCutter[1] [1], developed at the University of Maryland, is middleware used to explore and analyze scientific datasets stored on archival storage systems across a wide-area network. DataCutter provides a query-based interface with support for accessing subsets of datasets and for performing user-defined transformations of large data sets in archival storage. The processing structure is composed of a set of processes called "filters" that typically execute close to the data source. While the processing structure and data filtering ideas of the DataCutter are similar to ours, Armada is more flexible because it allows libraries to define the structure of distributed files and to provide an application-specific interface to the client application.

6 Summary

Efforts to develop high-performance computational grid applications have led to many new research challenges in the field of distributed computing. One of the greatest challenges is providing efficient I/O to grid applications. In a grid environment, where centralized control over system resources is not practical, we propose a flexible approach that allows application libraries to control the behavior and functionality of virtually all aspects of the file system. In Armada, applications access remote distributed files through a graph of application-defined objects called ships. The ships provide application-specific functionality near the data by executing on remote hosts known as harbors. Harbors provide the core file system functionality; they provide a secure environment that allows untrusted ships to access local resources, and they arbitrate among client ships competing for processor time, memory, disk access, and network access.

7 Status in February 2001

We are in the process of developing our first prototype of the Armada file system, and hope to produce experimental results within the next few months. Current efforts include developing tools for constructing XML-based ship graphs, a harbor implementation (built using J-Kernel [4]) to provide capability-based access to host resources, and a "fleet" of ships to allow construction of distributed files through a byte-level application interface. Initial experments will be conducted on a local cluster of Unix-based workstations.

[1]http://www.cs.umd.edu/projects/hpsl/Chaos.htm

References

[1] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the 2000 Mass Storage Systems Conference*, pages 119–133, College Park, MD, Mar. 2000. IEEE Computer Society Press.

[2] A. Colvin and T. H. Cormen. ViC*: A compiler for virtual-memory C*. In *Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '98)*, pages 23–33, Mar. 1998.

[3] E. Franke and M. Magee. Reducing data distribution bottlenecks by employing data visualization filters. In *Proceedings of the Eigth IEEE International Symposium on High Performance Distributed Computing*, pages 255–262, Redondo Beach, CA, Aug. 1999. IEEE Computer Society Press.

[4] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the 1998 Annual USENIX Technical Conference*, New Orleans, LA, June 1998.

[5] J. F. Karpovich, A. S. Grimshaw, and J. C. French. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, Portland, OR, Oct. 1994. ACM Press.

[6] D. Kotz. Expanding the potential for disk-directed I/O. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, pages 490–495, San Antonio, TX, Oct. 1995. IEEE Computer Society Press.

[7] D. Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[8] O. Krieger and M. Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 95–108, Philadelphia, May 1996. ACM Press.

[9] V. Messerli. *Tools for Parallel I/O and Compute Intensive Applications*. PhD thesis, École Polytechnique Fédérale de Lausanne, 1999. Thèse 1915.

[10] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assemply language. In *Proceedings of the Twenty-Fifth ACM Symposium on Principles of Programming Languages*, San Diego, CA, Jan. 1998.

[11] G. Necula. Proof-carrying code. In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997.

[12] G. Nelson, editor. *System Programming in Modula-3*. Prentice Hall, 1991.

[13] B. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–39, 1994.

[14] N. Nieuwejaar and D. Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

[15] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, February 1999.

[16] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The Netlogger methodology for high performance distributed systems performance analysis. In *Proceeding of IEEE High Performance Distributed Computing conference (HPDC-7)*, 1998.

[17] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Ashville, NC, 1993. ACM Press.

[18] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6), Oct. 1999.