CONF-950985--1

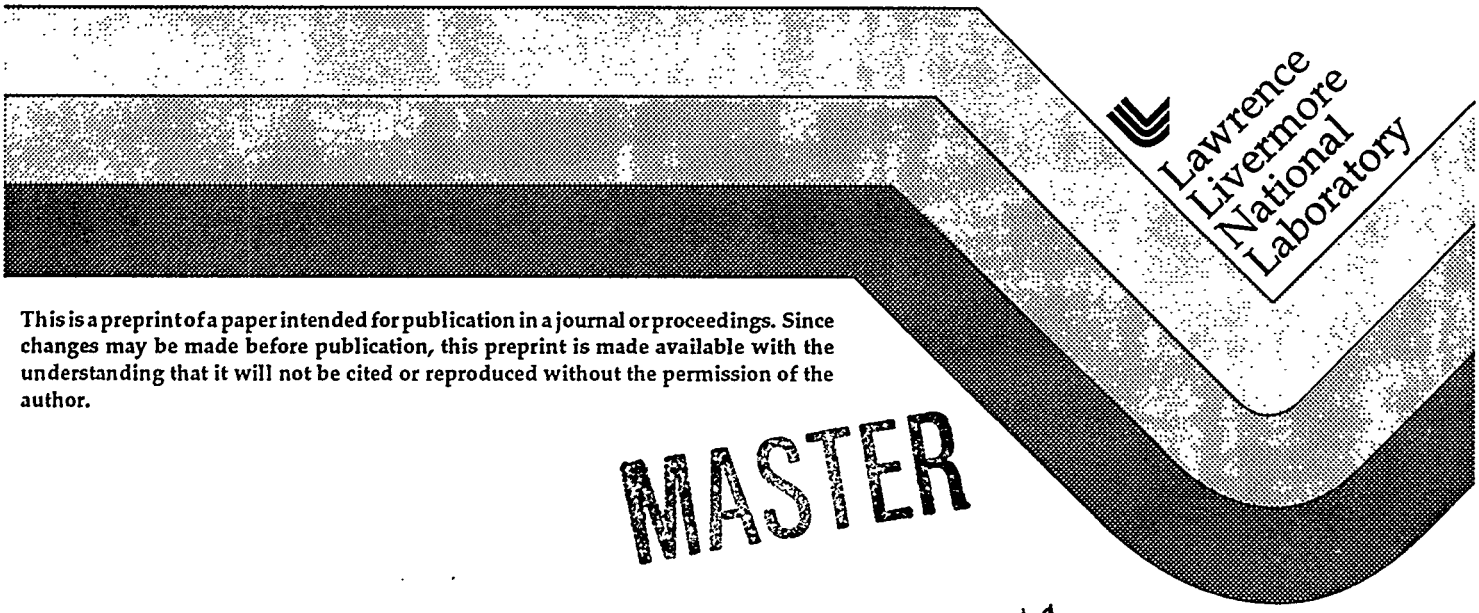
UCRL-JC-119772
PREPRINT

The Parallel I/O Architecture of the High Performance Storage System (HPSS)

R.W. Watson
R.A. Coyne

This paper was prepared for submittal to the
14th IEEE Symposium on Mass Storage Systems
Monterey, CA
September 11-14, 1995

February 1995



DISCLAIMER

DRAFT

The Parallel I/O Architecture of the High Performance Storage System (HPSS)

Richard W. Watson
Lawrence Livermore National Laboratory
Livermore, CA 94550

Robert A. Coyne
IBM U.S. Federal
Houston, TX 77058

Abstract

Rapid improvements in computational science, processing capability, main memory sizes, data collection devices, multimedia capabilities and integration of enterprise data are producing very large datasets (10s-100s of gigabytes to terabytes). This rapid growth of data has resulted in a serious imbalance in I/O and storage system performance and functionality. One promising approach to restoring balanced I/O and storage system performance is use of parallel data transfer techniques for client access to storage, device-to-device transfers, and remote file transfers. This paper describes the parallel I/O architecture and mechanisms, Parallel Transport Protocol, parallel FTP, and parallel client Application Programming Interface (API) used by the High Performance Storage System (HPSS). Parallel storage integration issues with a local parallel file system are also discussed.

Introduction

Rapid improvement in computational science, processing capability, main memory sizes, data collection devices, multimedia capabilities, and integration of enterprise data are producing very large datasets. These datasets range from tens to hundreds of gigabytes up to terabytes. In the near future, storage systems must manage total capacities, both distributed and at single sites, scalable into the petabyte range. We expect these large datasets and capacities to be common in high-performance and large-scale national information infrastructure scientific and commercial environments. One result of this rapid growth of data is a serious imbalance in I/O and storage system performance and functionality relative to application requirements and the capabilities of other system components.

To deal with these issues, the performance and capacity of large-scale storage systems must be improved by two orders of magnitude or more over what is available in the general or mass marketplace today, with corresponding

improvements in architecture and functionality. The goal of the HPSS collaboration is to provide such improvements. HPSS is the major development project within the National Storage Laboratory (NSL). The NSL was established to investigate, demonstrate, and commercialize new mass storage system architectures to meet the needs above [9,10,35]. The NSL and closely related projects involve more than 20 participating organizations from industry, Department of Energy (DOE) and other federal laboratories, universities, and National Science Foundation (NSF) supercomputer centers. The current HPSS development team consists of IBM U.S. Federal, four DOE laboratories (Lawrence Livermore, Los Alamos, Oak Ridge, and Sandia), Cornell University, and NASA Langley and Lewis Research Centers. Ampex, IBM, Maximum Strategy Inc., Network Systems Corp., PsiTech, Sony Precision Graphics, Storage Technology, and Zitel have supplied hardware in support of HPSS development and demonstration. Cray Research, Intel, IBM, and Meiko are cooperating in the development of high-performance access for supercomputers and MPP clients.

Architectural overview

The HPSS architecture is based on the IEEE Mass Storage Reference Model: version 5 [8,19] and is network-centered, including a high speed network for data transfer and a separate network for control (Figure 1) [6,9,18,24,28]. The control network uses the Open Software Foundation's (OSF) Distributed Computing Environment (DCE) Remote Procedure Call technology [29]. In actual implementation, the control and data transfer networks may be physically separate or shared. An important feature of HPSS is its support for both parallel and sequential input/output (I/O) and standard interfaces for communication between processors (parallel or otherwise) and storage devices. In typical use, clients direct a request for data to an HPSS server. The HPSS server directs the network-attached storage devices or servers to transfer data directly, sequentially or in parallel, to the client node(s) through the high speed data transfer network as shown in Figure 1. (HPSS also supports server attached devicesTCP/IP sockets and IPI-3 over High Performance Parallel Interface (HIPPI) are being utilized today; Fibre Channel Standard (FCS) with IPI-3 or SCSI, or Asynchronous Transfer Mode (ATM) will also be supported in the future [5,22,34,36]. Through its parallel storage and I/O support by data striping, HPSS will continue to scale upward as additional storage devices and controllers and network connectivity are added.

The HPSS components shown in Figure 2 can be distributed and multiprocessed and are multithreaded using DCE threads [29]. Multithreading is also important to serve large numbers of concurrent users. HPSS also uses the DCE security, distributed time and directory service services. HPSS uses the Transarc Encina software for support of atomic transactions, logging, and system metadata [14,25]. The storage devices managed by HPSS can be organized into multiple storage hierarchies [3,9]. Storage system management, built around an ISO managed object framework, is another important HPSS focus [4,21,23].

HPSS components can be run either on single or distributed server machines or on one or more nodes of a parallel machine. The former is expected to be common in environments where storage services are required for multiple client hosts. The latter case is likely to be used where very high speed storage integration is useful in a single parallel machine environment. An example is given later of such an arrangement using HPSS to support high speed parallel tape integrated with the IBM SPx disk based parallel file system.

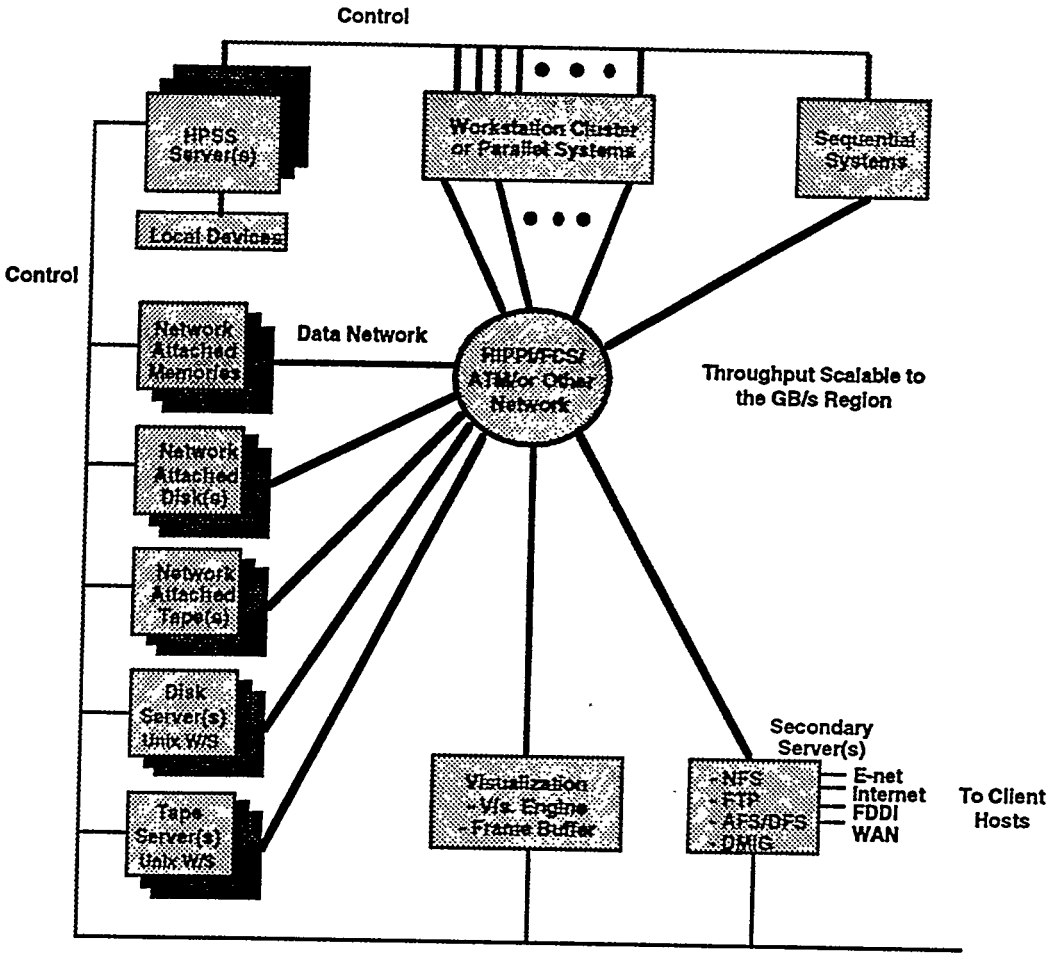


Figure 1. Example of the type of configuration HPSS is designed to support.

The key objectives of HPSS are:

- Scalability in several dimensions—for example, distribution and multiprocessing of servers, data transfer rates to gigabytes per second, storage capacity to petabytes, file sizes to terabytes, number of naming directories to millions, and hundreds to thousands of simultaneous clients.
- Modularity by building on the IEEE Reference Model architecture (see Figure 2) to support client access to all major system subcomponents, replacement of software components during the storage system's lifecycle, and integration of multivendor hardware and software storage components.
- Portability to many vendors' platforms by building on industry standards, such as the OSF DCE, standard communications protocols, C, POSIX, and UNIX with no kernel modifications. HPSS uses commercial

products for system infrastructure, and all HPSS component interfaces have been put in the public domain through the IEEE Storage System Standards Working Group.

- Reliability and recoverability through support for atomic transactions among distributed components, mirroring and logging of system metadata or user data, recovery from failed devices or media, reconnection logic, ability to relocate distributed components, and use of software engineering development practices.
- Client APIs to all major system components, including a parallel client API and interface to vendor parallel file systems, and support for industry standard services such as FTP (sequential and parallel) and NFS. Future support will include AFS/DFS, and Unix VFS. Support is also planned for interface to local and distributed file systems through the Data Management Interface Group (DMIG) standard.
- Security through DCE and POSIX security mechanisms, including authentication, access control lists, file permissions, and security labels.
- System manageability through a managed object reporting, monitoring, and database framework, and management operations with graphical user interface access and control.
- Support for better integration with data management systems through appropriate interface functionality at multiple levels in the architecture.
- Distributability by building on a client/server architecture and use of an OSF DCE infrastructure.

The shaded boxes for HPSS software components shown Figure 2 are defined in the IEEE Mass Storage Reference Model: version 5 [8,19].

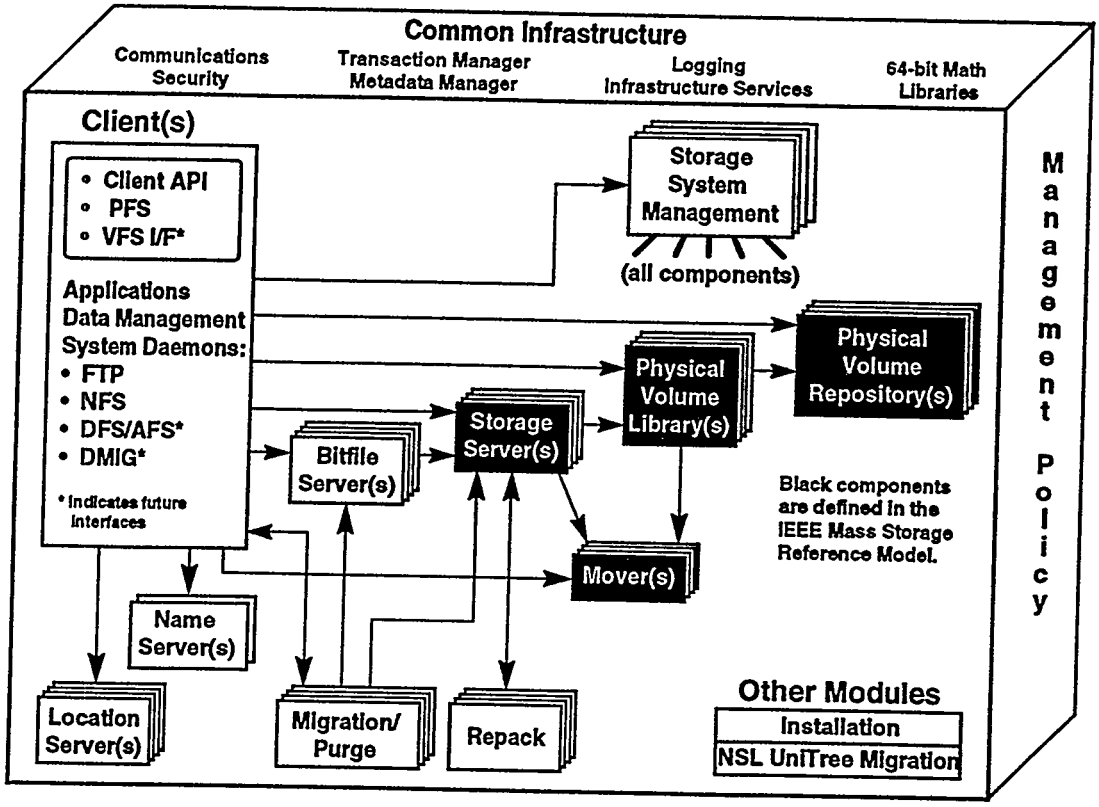


Figure 2. HPSS software model diagram.

The HPSS parallel I/O architecture

Parallel object model

Before we discuss the main components in Figure 2 and their role in HPSS parallel I/O we outline the model of parallel objects used in HPSS. Files are the main high level object discussed in this paper, but because all HPSS component APIs are accessible, many other types of objects can be built on lower level HPSS abstractions. A HPSS parallel file is logically a segment of bytes as in Unix. Two parameters specify how the parallel file is laid out on logical volumes (devices), the stripe width and the block size as shown in Figure 3.

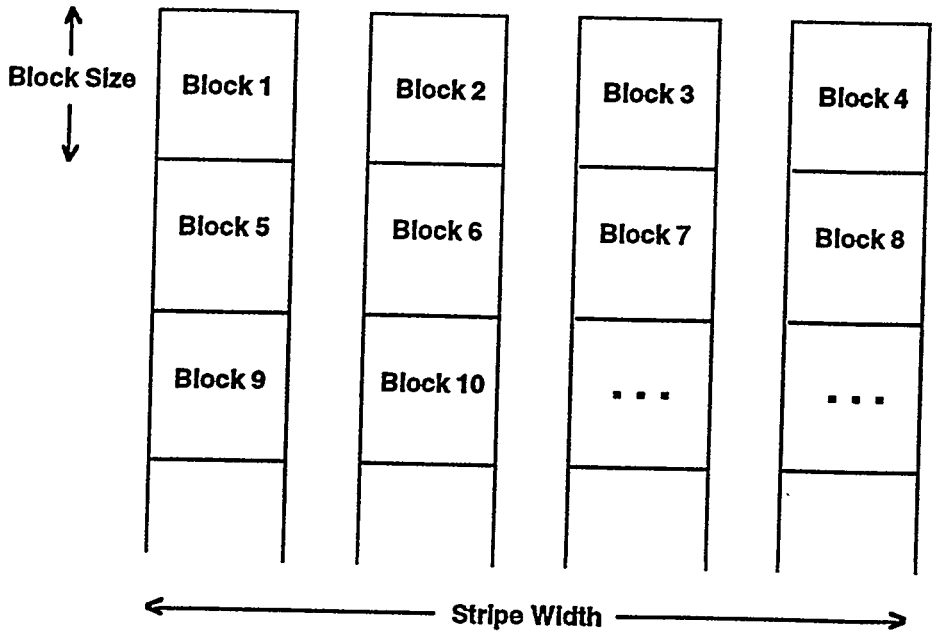


Figure 3. A file striped four ways on four virtual volumes (devices).

The stripe widths and block sizes available at each level within the many possible HPSS storage hierarchies [3], are established by the system administrator through the GUI Storage System Management interface [4]. For example, a given set of disks might be grouped in striped virtual volumes of widths 1, 2, 4 and 8. A set of tapes at the next level of the hierarchy might only be available in striped virtual volumes of widths 1, 2, and 4. This means that for migration both the 4 and 8 way striped disk files would migrate to 4 way striped virtual tape volumes in their respective storage hierarchies, but would cache back to disks as 4 or 8 way stripes as appropriate for their specified hierarchy. HPSS supports any stripe width, not just powers of two, although the latter is expected to be common. Setting the relative stripe widths and block sizes at different levels of a given hierarchy to be multiples of each other respectively will generally lead to increased parallel I/O and higher throughput.

When files are created, the storage hierarchy, stripe width and block size are specified either implicitly or explicitly by class-of-service (COS) parameters within the HPSS open/create function [26]. Implicitly the application might specify latency and bandwidth which are then mapped to an appropriate storage hierarchy and virtual volume parallelism by the Bitfile Server using data for such a mapping provided by the Storage Server. The appropriate mappings between the COS parameters and actual stripe width, block size and hierarchy specifications are set by the system administrator through the HPSS Storage System Management interface when virtual volumes are

created. Alternatively the application can explicitly specify the hierarchy, stripe width and block size, but these must conform to those available as configured by the system administrator.

The main design objectives of the HPSS parallel I/O architecture and mechanisms are listed in the Parallel Transport Protocol section below. Central concepts in the HPSS parallel I/O architecture are represented in the Parallel Transport Protocol (PTP). Following the PTP discussion we then discuss each of the HPSS components in Figure 2 and outline its role in HPSS parallel I/O. Following this discussion we provide several examples of parallel I/O supported by HPSS.

Parallel transport protocol

The Parallel Transport Protocol (PTP) used by HPSS has, during the past two years, influenced and been influenced by the work of an industry, government, university group interested in designing a protocol supporting parallel data exchange of datasets meeting the objectives listed below [1]. This protocol and the HPSS experience is also being used by the IEEE Storage System Standards Working Group P1244.Mvr standards group in its work [20]. PTP and HPSS parallel I/O design goals are:

1. Provide parallel data exchange between heterogeneous systems and devices (e.g., sources and sinks can be any distributed combination of storage abstractions such as segments of files, memory buffers in heterogeneous systems, physical devices, or network addresses.)
2. Support any combination of parallel and sequential sources or sinks.
3. Support network attached peripherals.
4. Support gather/scatter and random access across heterogeneous systems; in particular any combination of stripe widths, blocking factors and regular or irregular data blocks can exist on the source and sink sides of the transfer (i.e., no assumptions can be made about source and sink data layouts)
5. Provide I/O bandwidth improvements that implicitly scale with increasing physical parallel connectivity.
6. Maintain independence from the lower level transport protocol (i.e., the PTP should work with any type of network and transport protocol, TCP/IP, HIPPI, FCS, ATM, etc.).

7. Data flowing on any parallel path in a given parallel transfer should flow asynchronously to that on any other path.

8. Support efficient sequential I/O as a special case.

9. Support the separation of data and control needed for a flexible parallel I/O architecture.

The PTP sits above the Transport layer in a network architecture and can be used by a wide variety of higher level services and APIs. It is beyond the scope of this paper to give the PTP in detail. An outline of the PTP based on Reference [1] is now presented, with more detail in the Appendix. The general PTP configuration and network model is shown in Figure 4.

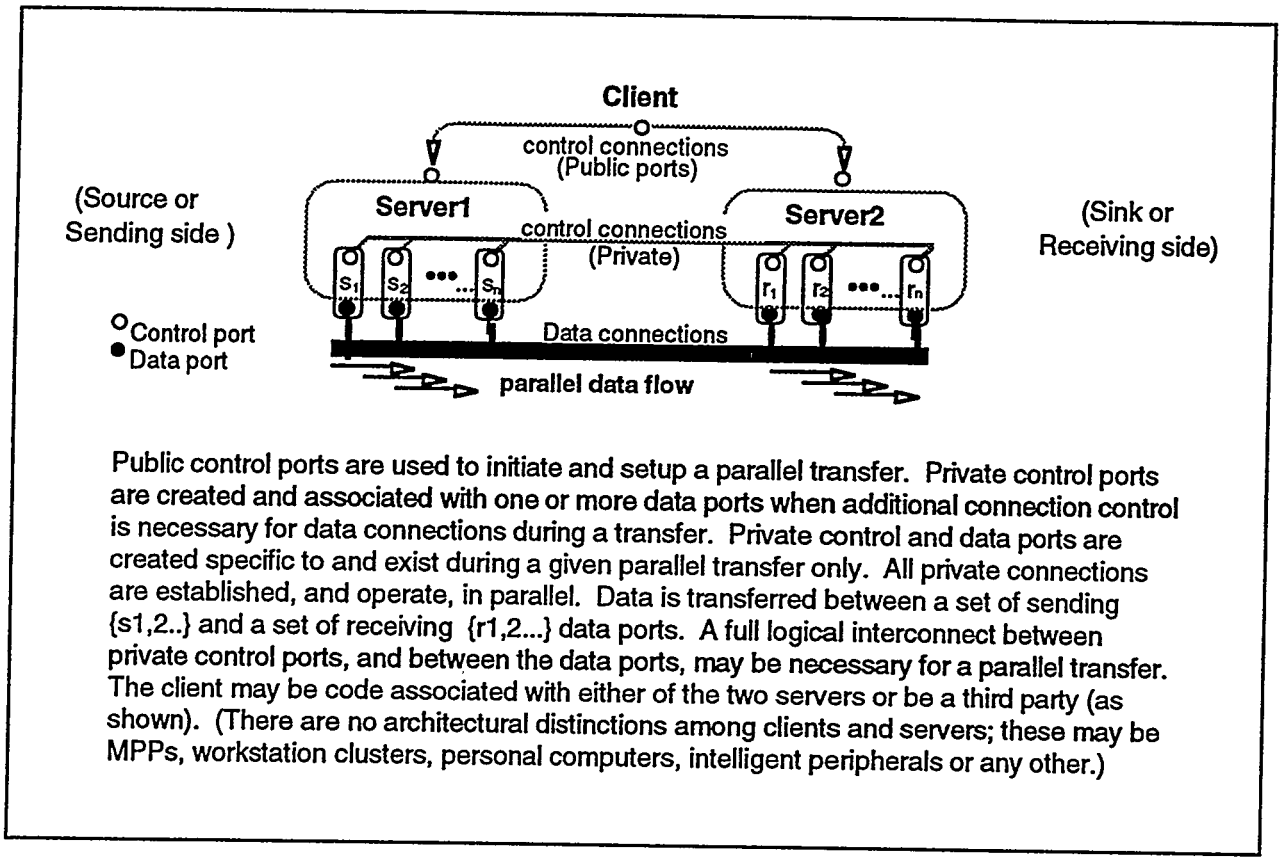


Figure 4. Parallel transport model.

The PTP model assumes three entities, the Client (that logical entity initiating a transfer), a Source Server (Server 1) (the source of data), and a Sink Server (Server 2, the sink of data). The Client could be a third-party or be a

module collocated at either the Source or Sink servers. The model is defined assuming connection mode transport communication services.

Two classes of connections, control and data, are defined as logically separate, although they may or may not share the same physical network(s). Control connections are characterized by small packets of requests and replies and data connections by large packets of data. A set of data connections are used for parallel transport of data. Implicit in a connection-oriented structure is the asymmetric structure that one side (the passive side) listens for connection requests and the other side (the active side) initiates connection establishment.

In general, the source and sink data may be distributed over any of the abstractions listed in the first design goal and be located anywhere in the network. Thus, the Source and Sink servers shown in Figure 4 may be recursively made up of many distributed cooperating entities.

The PTP defines: (1) the data structures that specify the distributed Source data (gather-list) and the distributed Sink data (scatter-list); (2) the logical mapping mechanism between the gather/scatter lists; (3) the control information that must be exchanged among Client, Source and Sinks to control the transfer; and (4) the mechanism to specify the detailed transfer plan of required connections and which data flows over which connection.

The data structure that specifies the gather/scatter-lists in requests among clients and servers is called an I/O Descriptor (IOD). The corresponding data structure used in replies among servers and clients is called an I/O Reply (IOR). It should be noted that in an actual implementation, such as in HPSS, a server at one level can be a client of another server at a lower level.

The IOD data structure contains the following information:

- function (e.g., read or write)
- source-descriptor-list (the gather-list of data source locations)
- sink-descriptor-list (the scatter-list of data sink locations)
- a unique transfer ID generated by the client code

The IOR record contains the following information:

- a unique transfer ID generated by the client code.
- a set of flags indicating successful or unsuccessful completion and an error code.
- source-reply-list, a description specifying source data results

- sink-reply-list, a description specifying sink data results.

The concept introduced to support the mappings from the data locations specified in the gather-list of source-descriptors, to the data locations specified in the scatter-list of sink-descriptors is called the *transfer window* (or *window*). The window is viewed as containing the n logically contiguous bytes of the total transfer. Each source-descriptor in the gather-list specifies an offset and length of m unique bytes (where m may be different for each descriptor) for where the source data described is to be placed in the window and each sink-descriptor in the scatter-list specifies an offset and length of k unique bytes (where k may be different for each descriptor) for where the sink data described is to be obtained data from the window. The source/sink descriptors also contain an *address* structure which define the location of the actual data. This address structure can take several forms. For example, it might be a network address (port); a file ID, offset and length; or a stripe specification. The stripe specification contains an address (e.g., device, memory buffer, port) and an offset where to begin to get or put data from/to the device. There is also a block size and a stride (i.e., the number of blocks to skip between blocks) defining how to get or put data from/to the window to/from the devices.

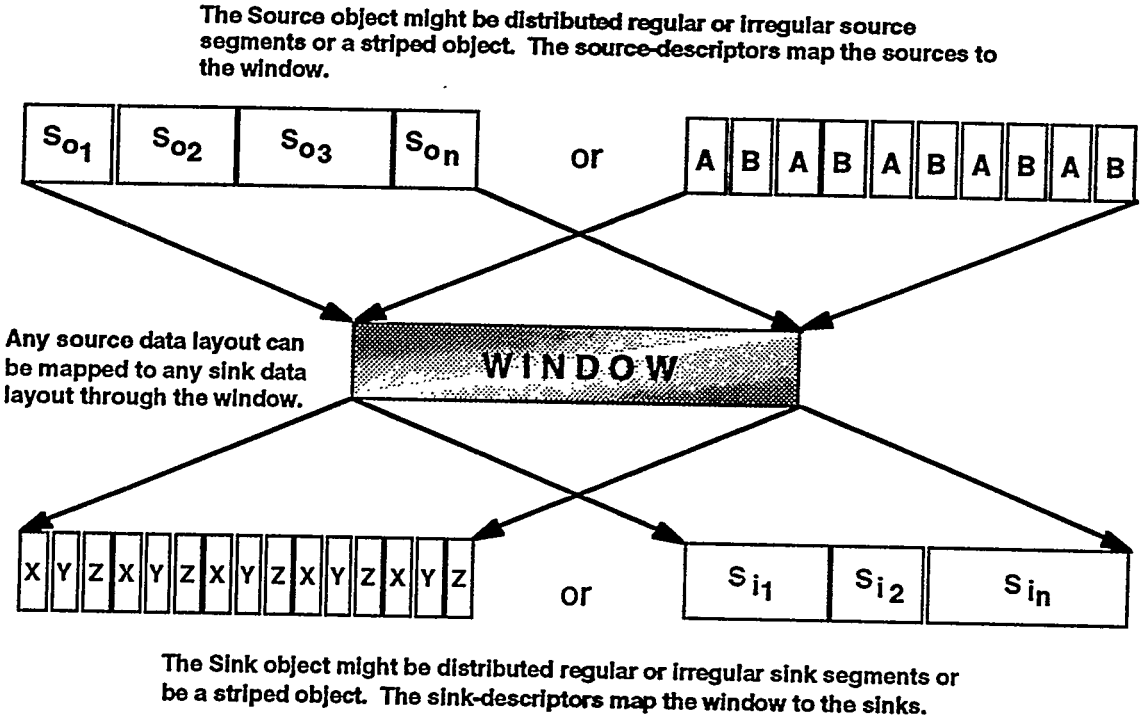


Figure 5. Mapping between gather-list and scatter-list using a logical window.

A user or application request to a Client initiates a transfer. The Client might be in a parallel I/O library or a file transfer client module as shown in later examples. Well known network control addresses (*control ports*) are assumed to be publicly available for the Client and Source and Sink servers. Public control connections are used to initiate, control, and describe the transfer; these are setup exchanges which are sequential. Private control connections may be established during the transfer as necessary to control transfers on specific data connections. Data is moved in parallel over sets of data connections between storage objects in the Source to storage objects in the Sink. Data connections are private (i.e., created specific to a transfer).

The basic mechanics underlying PTP are quite simple as follows:

1. The Client connects to the well-known control port of the passive server and sends a unique transfer ID and a message to move data. This message contains the IOD (which contains the function (read or write)) described earlier to specify the gather/scatter lists and window mappings. Either the Source or Sink can initially be the passive server. If the Client is collocated with one of the servers, the server is initially made passive to save control exchanges. During control exchanges between Movers at the start of the data transfer phase, the passive/active roles are renegotiated. The passive server replies to this request with an IOR containing a set of descriptors with ports (network addresses) to use for the transfer and their relationship to the data described in the IOD. These ports, depending on the lower level protocol in use, are either control ports to be used to establish and control the data connections, or the actual data ports to use for the transfer. All passive ports then listen for connect requests from the active server.
2. The Client connects to the well-known control port of the active server, sends the unique transfer ID, and IOD structure which includes the passive server's source or sink descriptors with the ports returned above.
3. Private control ports set up data ports or data ports have been defined directly. The active data ports connect (in parallel) to the listening passive data ports and transfer data in parallel. For a detailed description of how data is transferred on a single connection see Reference [18].
4. The Servers reply to the Client with completion status in IORs.

Each Server manages the movement of data between its physical devices and data ports. The protocol does not introduce dependencies on how data is distributed within the server (this distribution is architecturally specific to both a given system and the hardware involved).

With the above introductory description as background, the reader can now either continue on with the main body of the paper or go to the Appendix where we give a more detailed outline of PTP. We now outline each component in Figure 2 and discuss its role in HPSS and parallel I/O. In particular, we discuss how the IOD that describes the I/O is formed.

HPSS component roles in parallel I/O

Application Interfaces. HPSS supports several high-level interfaces: currently Client API, FTP (both standard and parallel), and NFS; with DFS/AFS, and VFS planned for future releases. We say a few words on those currently supported here and then illustrate their support of parallel I/O and parallel files later. While not a client API, we also plan to support the DMIG specification for integration with other file systems. All the client interfaces contain control code to form IODs and communicate with HPSS. They also contain client mover code to perform the parallel I.O.

Client API. The HPSS Client-to-file server API mirrors the POSIX file system interface specification where possible [33]. The Client API also supports extensions to allow the programmer to take advantage of the specific features provided by HPSS (e.g., class-of-service passed at file creation and support for parallel data transfers) [33]. The HPSS client API supports client access to HPSS files as well as integration with other services such as FTP, NFS or a local parallel file system. The client API code contains the Client and passive server defined in the PTP. In parallel I/O, the key functions are the HPSS open/create function discussed earlier which determines the storage hierarchy, stripe width and block size of the file; standard POSIX HPSS read and write functions which implicitly support reading or writing a parallel file into or out of a single buffer or buffer list; and HPSS readlist and writelist functions which support explicit specification of multiple sinks and/or sources of data using the IOD data structure discussed earlier. The HPSS read, write, readlist and writelist client functions are mapped by the client API code into Bitfile Server reads and writes which take the IOD as an argument and returns the IOR structure discussed earlier. The client API code also contains Movers (see below) to manage its half of the data transfer phase of the Parallel Transfer

Protocol. Asynchronous I/O, data caching/buffering, and collective XXXXXXXX in which many processes perform I/O as a group can be provided by higher level functions, if desired.

FTP (standard and parallel). HPSS provides a standard FTP server interface to transfer files between HPSS and a local file system. In addition, a Parallel FTP (PFTP), an extension and superset of standard FTP, has been implemented to provide high performance data transfers between client systems and HPSS via parallel data paths.

NFS. The NFS V2 Server interface for HPSS provides transparent access to HPSS name space objects. NFS only supports sequential transfers to the clients. Our goal in the NFS server design and implementation is to provide the same performance (throughput and NFS transaction rate) as the native O.S. NFS on which the HPSS NFS server runs. Client systems to both the native HPSS and the NFS V2 service see the same name space and data. The NFS V2 Server caches data from HPSS in large blocks at high speed using third-party, parallel data paths.

Parallel file system. HPSS provides the capability to act as an external hierarchical file system to vendor Parallel File Systems (PFS). The first PFS/HPSS integration, discussed later, supports the IBM SPx PIOFS. Early deployment is also planned for Intel Paragon, Meiko CS-2 PFS and IBM RS 6000 cluster integrations with HPSS.

Name server (NS). The NS maps a user POSIX path name to an HPSS object. The NS provides a POSIX view of the name space, which is a hierarchical structure consisting of directories, files, and links. Namable objects are any object identified by HPSS Storage Object IDs, such as sequential or parallel files or parallel virtual volumes. No knowledge of parallel I/O is required of NS. The NS also provides the POSIX file protection services.

Bitfile server (BFS). The BFS provides the POSIX byte stream file abstraction to its clients, supporting bitfile sizes up to 2**64 bytes. Bitfiles can be striped as illustrated in Figure 3. The BFS supports both sequential and parallel read and write of data from/to bitfiles. Sequential I/O is an efficient special case of parallel I/O with stripe width of one. Parallel files with a given stripe width and block size are created during the HPSS open/create call as described earlier.

The reads and writes to a file can be random, with the system managing reads and writes of partial files. There are two flavors of reads and writes. The first flavor is a standard POSIX read or write with the parallelism and blocking into or out of the specified buffer(s) being handled implicitly by the system. The second flavor is a form of read and write, called readlist and writelist, wherein the application, using IODs, can explicitly define the data layout of sources or sinks and mappings from these layouts to a striped HPSS file. An application buffer layout could, for example, be on different nodes of an SMP or MPP, or different systems in a distributed set of workstations or other systems, or on a workstation cluster. The data layout can be regular or irregular contiguous chunks or be striped across nodes. Similarly, the IOD can specify arbitrary layouts on an HPSS file.

The client API parallel I/O support code (e.g., library) provides for control messages to and from the Name Server for mapping file path names to bitfile-IDs and to and from the Bitfile Server for the BFS_Read, and BFS_Write functions. The two control parameters exchanged between the client API library and the BFS are the I/O descriptor (IOD) and the I/O reply (IOR) described earlier in the Parallel Transport Protocol description.

Within the scope of this paper we only use the IOD in our examples, IORs are assumed. One of the central concepts of the HPSS parallel I/O architecture is the specification of the data transfer plan through successive mappings of that part of the IOD structure for which HPSS is responsible as it is passed through HPSS's layered Bitfile Server, Storage Server and Mover abstractions. The client API code provides the necessary detailed information in the IOD for the part of the I/O for which it is responsible. Each server only has to know as much about the parallel I/O as is necessary for its role. The data transfer plan and mappings to and from the transfer window, outlined in the PTP discussion, are thus developed as the IOD(s) travels through the HPSS servers.

The IODs may start out at the application level as very simple, general parameters or be quite explicit, but are, in either case, successively mapped as appropriate into more detailed IODs by each HPSS server in the HPSS-layered abstractions until ultimately, at the Mover level, IODs contain explicit communication ports, device addresses, offsets and lengths and their relationships to window offsets and lengths for the complete PTP parallel data transfer. Thus, the parallel transfer plan is the result of the transformation algorithms applied to the IOD as it passes through the system. For example, for a write function the source and sink-descriptor-lists of an IOD received by the BFS might consist of source-descriptors specifying window offsets and lengths and ports that have been mapped to source buffer addresses and lengths, and a sink-descriptor that has a window offset and length and a single bitfile ID, offset,

and length. The BFS then maps the single sink descriptor with the bitfile ID, offset and length into multiple sink-descriptors for each storage segment with appropriate window offsets and lengths and storage segment IDs, offsets and lengths and passes the expanded IOD structure to the Storage Server (SS). The detailed source-descriptor-list part of the IOD obtained from the client is passed on unchanged through the servers in the write example. The SS then maps the sink-descriptors with storage segment IDs, offsets and in lengths into sink-descriptors with device IDs, offsets, and lengths, and appropriate window offsets and lengths, and passes the expanded IOD structure to the Mover. The Mover carries out the parallel I/O transfer plan defined by the fully developed IODs. Later we give several examples of HPSS parallel I/O. The BFS deals only with the storage segment abstraction provided by the Storage Server and is itself largely unaware of the parallel I/O mechanisms.

Storage server (SS). The SS provides a hierarchy of storage object abstractions: logical storage segments, virtual volumes, and physical volumes. All three layers of the SS can be accessed by appropriately privileged clients. For example, a parallel DBMS could be implemented directly on the SS storage segment layer. Our discussion in this paper uses the bitfile abstraction for all examples. The SS translates references to storage segments into references to the corresponding virtual volumes and finally into physical volume references and device addresses. It also schedules the mounting and dismounting of removable media through the Physical Volume Library. An important simplifying SS design assumption is that storage segments do not span virtual volumes. The SS, in conjunction with the Mover, has the main responsibility for orchestration of HPSS's parallel I/O operations. The SS is responsible for all virtual and physical volume allocations.

The SS receives storage segment IDs, offsets and lengths in IODs from the BFS and maps them first to the appropriate virtual volume IDs, offsets and lengths, then to physical volume IDs and offsets and length, specifying during this process appropriate window offsets and lengths. The SS then issues an atomic mount to the Physical Volume Library [12] component to mount the physical volumes (tapes) in the virtual volume. Atomic mounts are used as part of the deadlock avoidance mechanism [12]. After the mount is completed, the SS forks off multiple parallel threads to handle the I/O for each physical volume and associated I/O device. The SS threads then pass on the expanded IOD structures to the Movers associated with the specified physical devices to carry out the data transfer specified by the IODs.

Physical volume library (PVL). The PVL manages all HPSS physical volumes. Clients, such as the SS can ask the PVL to mount and dismount sets of physical volumes. Clients can also query the status and characteristics of physical volumes. The PVL maintains a mapping of physical volume to cartridge and a mapping of cartridge to PVR and PVR location. The PVL also controls all allocation of drives. The PVL does not understand the concept of parallel I/O and only executes the requested atomic or non-atomic physical volume mounts. All volume mount requests from all clients are handled by the PVL. This allows the PVL to prevent multiple clients from deadlocking when trying to mount intersecting sets of volumes [12]. The standard HPSS mount interface is asynchronous, that is multiple mounts can be going on concurrently.

Physical volume repository (PVR). The PVR manages all HPSS supported robotics devices and their media such as cartridges. Clients, such as the PVL, can ask the PVR to mount and dismount cartridges. Every cartridge in HPSS must be managed by exactly one PVR. Clients can also query the status and characteristics of cartridges. The PVR consists of these major parts: Generic PVR service, and support for devices such as Ampex, STK, and 3494/3495 robot services, as well as an operator mounted device service. The PVRs contain no knowledge of parallel I/O.

Mover (Mvr). Movers are responsible for transferring data from source devices to sink devices. A device can be a standard I/O device with geometry (e.g., a tape or disk), or a device without geometry (e.g., network, memory). There are Movers for each type of device and network. Movers also perform device control operations. Movers perform the control and transfer for both sequential and parallel data transfers. The Movers, along with the Storage Servers, are the entities that are primarily responsible for HPSS parallel I/O. The Movers perform the data transfer part of the Parallel Transport Protocol outlined earlier, using the appropriate network transport protocols as determined by the source(s) and sink(s) connectivity. For example, this might be parallel transfers using IPI-3 on HIPPI, or Socket TCP/IP connections. The Movers determine the final data transfer plan of what data in what sequence goes over what required connections using the information in the detailed IODs they receive from the SS

threads. There are Movers on both the client and HPSS sides of a transfer with essentially identical code and functionality to implement their roles in the PTP.

Storage system management (SSM). The HPSS SSM architecture is based on the ISO managed object architecture [21,23]. The Storage System Manager (SSM) monitors and controls the available resources of the HPSS storage system in ways that conform to the particular management policies of a given site [4,33]. With respect to parallel I/O we have already mentioned the important role of SSM in providing the system administrator interface to control the configuration of the striping and blocking factors at each level of the storage hierarchies, the establishment of multiple hierarchies, and the grouping of physical volumes such as disks and tapes into striped virtual volumes. It also supports the system administrator in establishing the mapping of class-of-service parameters to virtual volume and other I/O characteristics. Setting up the striping and blocking at each level of a hierarchy to be multiples of each other is an example of the care the system administrator must exercise to achieve optimal performance. Managing stripe widths for tape under various load assumptions is also important [17].

Migration and caching. Automatic migration and caching are controlled by a Migration/Purge server within storage hierarchies using the appropriate bitfile and storage servers. HPSS also supports explicit HPSS also supports explicitly Storage and Purge commands in the client API. Purge first migrates the specified data, if necessary, before deleting it. As stated earlier the system administrator uses the GUI interface and Storage System Management services to organize the devices being managed by HPSS into storage hierarchies consisting of virtual volumes at each level with specified characteristics. The stripe width and block size of the virtual volumes at each level can be of any size and the Movers will perform the appropriate blocking or deblocking and data transfers as storage objects such as files are migrated down the hierarchy from faster more expensive devices to slower less expensive devices, or cached up the hierarchy in the opposite direction. Later we give an example of the type of device-to-device I/O performed during migration and caching. Caching of partial files to the top of the hierarchy is triggered by an access or stage request. When the block containing the requested data arrives, it is delivered to the rquestor without waiting for the entire cache or storage to complete. File migration is triggered by higher level volumes filling up. Choice of parts of files to migrate is based on parameters such as size and access histroy.

Examples of HPSS parallel I/O

This section provides parallel I/O examples directly using the Client API library, device-to-device copy, parallel I/O for caching and migration to/from the NFS server, and Parallel FTP. The first two examples are given in some detail to illustrate the concepts and mechanisms presented in earlier sections. The remaining examples are given with less detail to give a general sense of the parallel I/O control and data flows. Implicit in all the examples is the mapping to or from window offsets and lengths that is taking place as the IODs are handled by each server.

Application Client Parallel API

As an example of an application client to HPSS parallel data transfer and the PTP mechanism, consider the four node parallel HPSS read of a disk file diagrammed in Figure 6. The Client in the PTP description is a module within the client API library code. Similarly, for this example, the Sink Server and associated client Movers of the PTP description are also within the client API library code. The Source Server in this example is HPSS. In somewhat simplified form, with corresponding message numbers on Figure 6, the following communications take place.

1. The application opens the HPSS file with a call to the client API code which sends a message to the Name Server and receives a bitfile ID in return. It then calls the client API code to open the file which in turn calls the BFS and receives a file-descriptor. The application issues a readlist function to the client API code with an IOD that in the source-descriptor-list has the file-descriptor, offset and length and in sink-descriptor-list has the machine IDs and buffer addresses of the application machine's four sink nodes.
2. The client API library code logically communicates with its collocated Sink Server code, and using the information in the IOR returned, maps the IOD sink-descriptor-list machine ID and buffer addresses to port IDs and issues a BFS_Read call to HPSS with the expanded IOD. HPSS in this case is the active server.
3. The BFS then modifies the IOD's source-descriptor-list, which is a file-descriptor, offset, length, mapping file-descriptor, offset, length to corresponding storage segment, offset, length source descriptors for this file and issues a SS_Read with the expanded IOD.

4 The SS maps the segment IDs, offsets and lengths to virtual volume IDs, offsets and lengths, then creates physical volume IODs for each physical volume, forks threads, one for each physical volume, to perform the corresponding I/O for a IOD. The threads map the source-descriptors in the IODs to device addresses, offsets and lengths and issue Mvr-Reads to the Movers corresponding to the devices.

5. The Movers, in parallel, using the information in the completely specified IOD each receives and the unique transfer ID (TID) establishes connections to the listening application client Movers. The data transfer protocol on each connection contains a header with a TID, offset and length along with the data so that the data on each connection can be sent asynchronously in parallel and be properly placed in the client buffers [18]. The HPSS/client Mover pairs do the data transfers in parallel.

6. The client Movers store the data in the client buffers.

7. The Movers return IORs up their respective chain containing completion status.

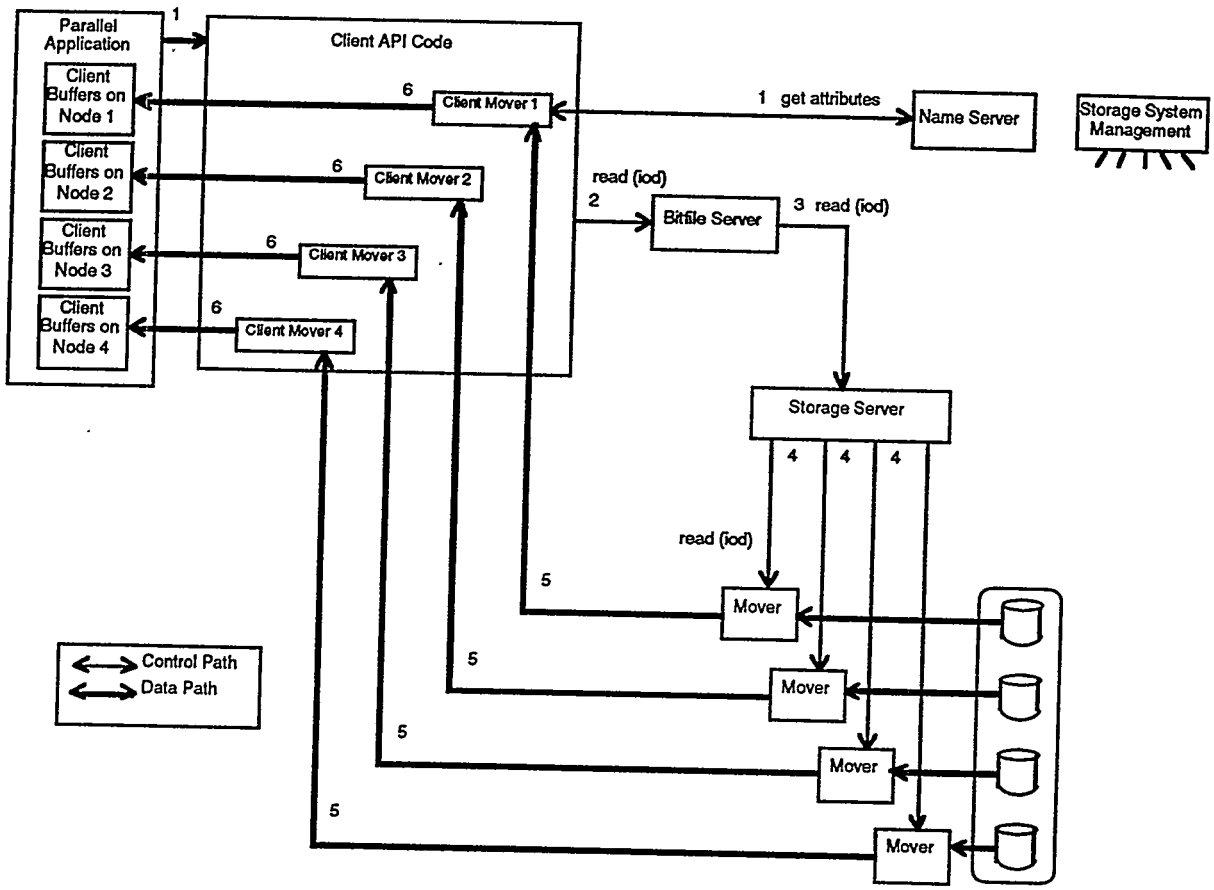


Figure 6. Client parallel transfer.

The above description would have had some additional steps if the read was from parallel tape. In that case, the tape Storage Server would have sent an atomic mount request to the PVL which in turn would have sent mount requests to the PVR. When the atomic mount operation was complete the PVL would have notified the tape Mover to read the tape labels and then replied to the Storage Server. The Storage Server would then have sent read requests and IODs to the Movers as before. The tape Mover would then have positioned the tapes before initiating the data movement.

HPSS Device-to-Device Parallel Copy

We now give a more complicated example showing the use of the PTP in a device-to-device transfer within HPSS. The particular example is a parallel file copy, but the central data transfer mechanism would also be used during migration and caching of data between devices at different levels of the storage hierarchy. In this example, the Bitfile Server is the PTP Client and the tape and disk Storage Servers and their associate Movers are the PTP Sink and Source servers respectively.

Figure 7 illustrates the control and data flow for the operation. Within the boxes are the HPSS component subsystems. The multiple Mover threads are shown but not the multiple SS threads. The numbers indicate the message flow sequence. In the control flow depicted, data is being copied in parallel from disk to tape.

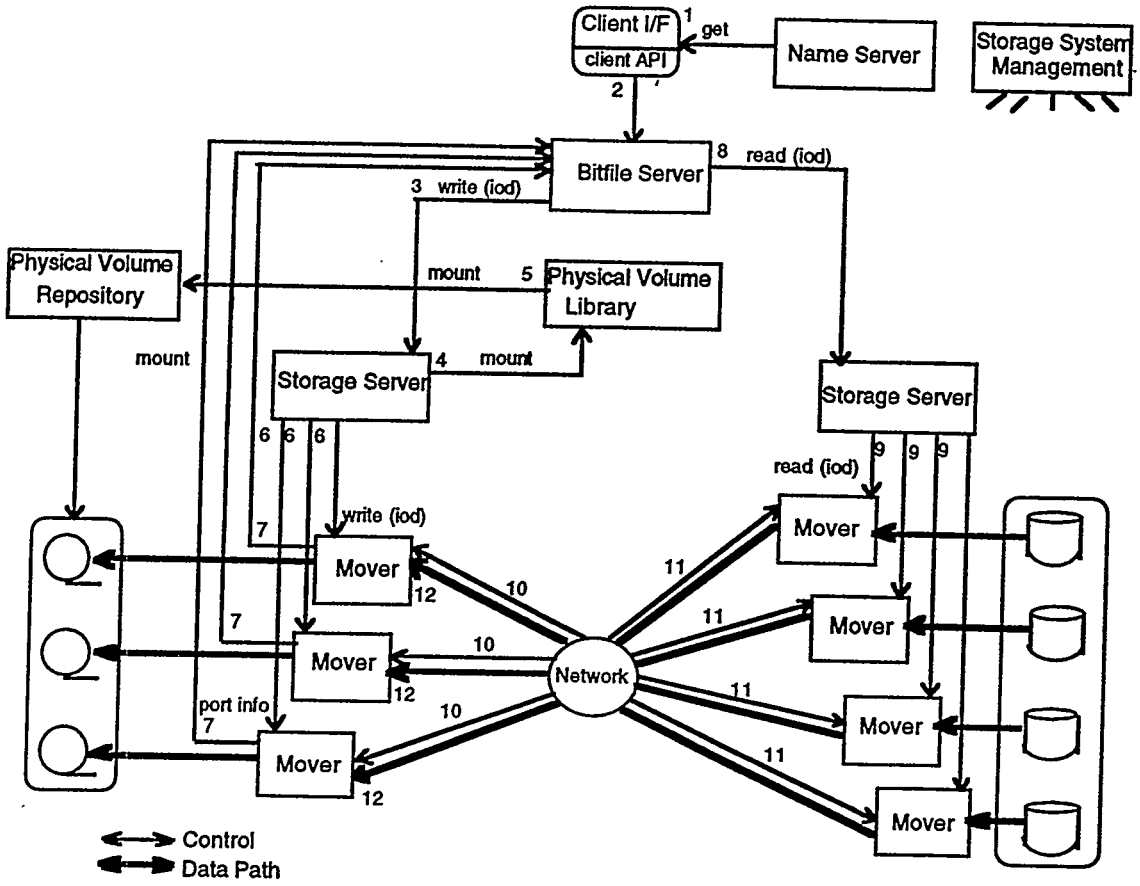


Figure 7. HPSS parallel-to-parallel transfer.

In the example depicted in Figure 7, a parallel disk-to--tape copy of a file is being performed. The file on disk is striped across 4 devices, and is being transferred as a striped file to 3 tape devices. The steps in performing the parallel transfer are listed below. The steps are split into the following categories, open, data movement, and close.

Control flow for disk file open:

- A client calls the HPSS open API to open the disk file being transferred.
- The open API issues a get attributes request to the Name Server to retrieve the bitfile ID corresponding to the name of the file being transferred. The external file name is a POSIX file path name.
- The open API issues an open call to the BFS, obtaining a file-descriptor.

Control flow for tape file open:

- A client calls the HPSS open API to open (and create) the tape file. The external name is a POSIX file name.

- The open API issues a create request to the Bitfile Server.
- The Bitfile Server returns the bitfile ID to the open API.
- The open API issues an insert request to the Name Server to associate the external file name with the bitfile ID returned.
- The open API maps the bitfile ID to a POSIX file-descriptor.

Control flow for data movement:

- The client calls the HPSS writelist API to copy the file from disk to tape. An IOD with source/sink file descriptors, offset, length and window offset and length information is supplied.
- The writelist API issues a write request to the Bitfile Server indicating the source/sink descriptor information. The IOD is modified to pass bitfile source/sink information to the Bitfile Server.
- The Bitfile Server translates the request into a logical segment and issues a create segment request to the tape Storage Server.
- The tape Storage Server returns a segment ID to the Bitfile Server. As additional storage segments are required during the transfer, this and the previous steps are repeated.
- The Bitfile Server issues a write request to the tape Storage Server. The IOD is modified to contain storage segment sink information used by the Storage Server.
- The tape Storage Server translates the storage segment references to virtual volume references, and then to the physical volumes associated with the virtual volumes then SS threads are forked to handle the physical volume I/O.
- The tape Storage Server locates Movers associated with the physical devices for the parallel transfer.
- The tape Storage Server issues an atomic mount request to the Physical Volume Library to atomically mount the tape physical volumes.
- The Physical Volume Library issues mount requests to the Physical Volume Repository which manages the tape media type.
- The Physical Volume Library issues a request to the tape Movers to read the tape labels to assure the correct tapes were mounted.

- The Physical Volume Library returns the device address of the tape read/write station (sink information) to the tape Storage Server.
- The tape Storage Server issues a write request to each of the tape Movers. The IOD has been modified to pass device sink information to the tape Movers.
- Each tape Mover returns its listen-port addresses to the Bitfile Server in an I/O reply.
- The Bitfile Server issues a read request to the disk Storage Server. The tape listen-port addresses are passed in the IOD sink information.
- The disk Storage Server translates the storage segment references to virtual volume references, and then to the physical volumes associated with the virtual volumes.
- The disk Storage Server determines which disk Movers are associated with the parallel transfer.
- The disk Storage Server forks off threads for each disk volume and in turn issues a read request to a corresponding disk Mover. The IOD has been modified to pass device source information to the disk Movers.
- The disk Movers connect to the tape Mover listen-ports, and send transport options and stripe and blocking information.
- The tape Movers respond with data port addresses and transfer information (during the transfer, the tape Movers will be the active entities).
- The disk Movers connect to the tape Mover data ports and transfer data.
- Once all data is transferred, the disk and tape Movers send IOR to the disk and tape Storage Server threads, which in turn, send IORs to the Bitfile Server.
- The Bitfile Server replies to the client.

Control flow for disk close:

- The client calls the HPSS close API to close the disk file.
- The close API issues a close request to the Bitfile Server to close the bitfile.

Control flow for tape close:

- The client calls the HPSS close API to close the tape file.
- The close API issues a close request to the Bitfile Server to close the bitfile.

- The Bitfile Server issues a (storage segment) unmount request to the tape Storage Server.
- The tape Storage Server translates the storage segment references to virtual volume references, and then to the physical volumes associated with the virtual volume. A request is then issued to the Physical Volume Library to unmount the physical volumes.
- The Physical Volume Library issues unmount requests to the Physical Volume Repository to actually unmount the tape cartridges.

NFS Transfer

Figure 8 depicts the components of the NFS V2 Server [31]and their interrelationships. This particular example is for an NFS read operation. The numbers indicate flow sequence. The NFS Mount daemon, which runs as a separate process from the NFS server, reviews the export list to determine if the client may mount the directory. If the mount request is honored, the Mount daemon returns an NFS handle which is used in subsequent requests to the NFS server to obtain NFS handles to HPSS file objects.

NFS clients access the NFS server APIs through the reentrant ONC RPC library. When the NFS server is requested to operate on an HPSS file object, the NFS handle that is sent in the request is verified with the export list.

Requests that may time-out, causing the client to reissue the request, are saved in a structure. This structure is queried for requests that are likely to cause duplicates. If a request is determined to be a duplicate request, the NFS server sends the reply data that is also saved in the structure.

The client API library provides the control path to HPSS. Both HPSS and local control information are cached in structures, which are managed by the header cache and data cache libraries, respectively.

When a bitfile is read or written, the data is read from or written to the NFS data cache. Bitfile data is retrieved from (written to) HPSS when required by the data cache manager, which provides the high speed parallel data path to HPSS. The parallel I/O is handled as previous examples have outlined.

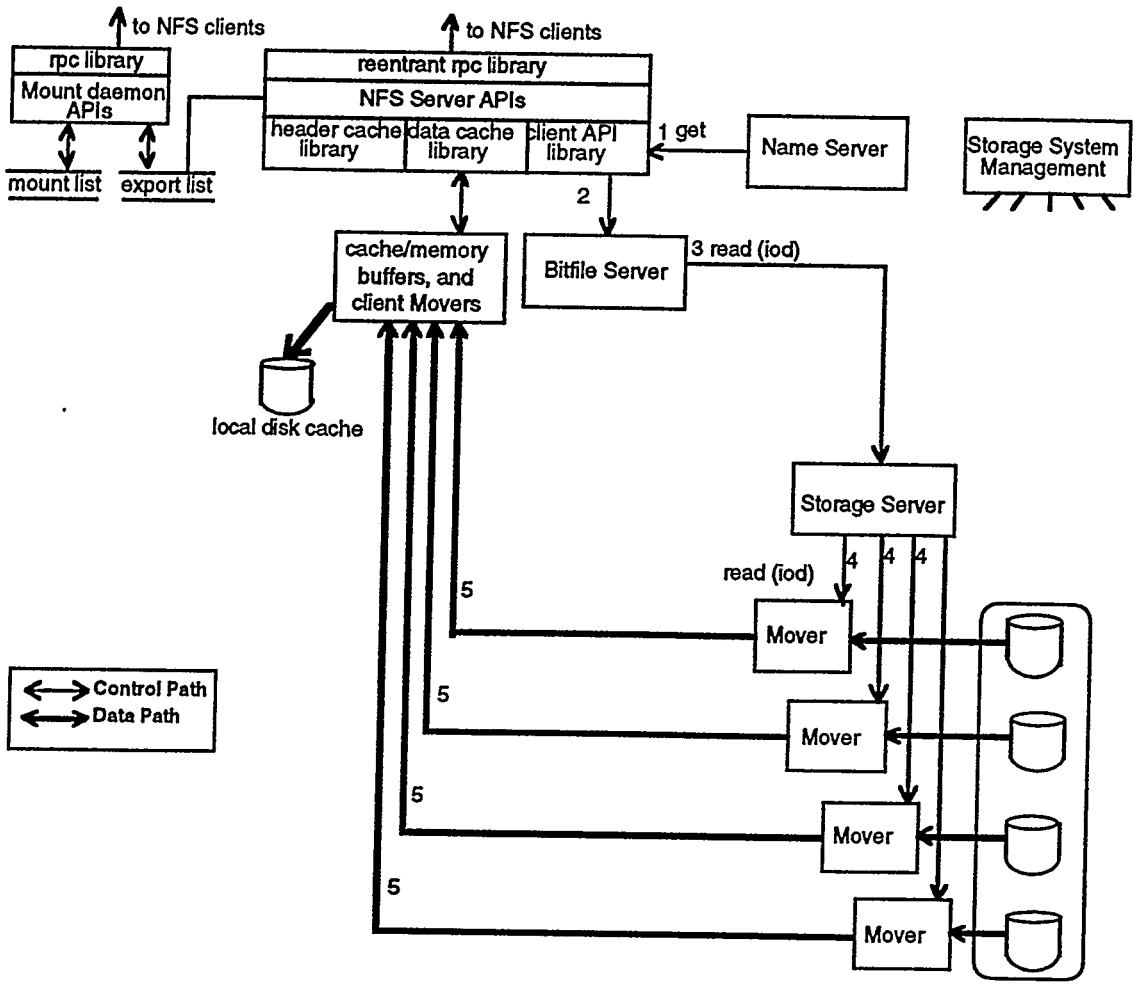


Figure 8. NFS transfer.

Parallel FTP (PFTP)

The standard Internet FTP [22] supports sequential transfer (get, put, append) of whole files. It supports separation of data and control connections and, therefore, it supports control requests and replies on one network and data transfers on another network (e.g., higher speed network) or enables a client on one host to initiate a transfer between a source and sink on two other hosts. This separation of control and data transfers is exactly the architecture needed to support high speed FTP service for HPSS. What is needed are extensions to the standard FTP to support parallel files and parallel transfers [32]. The FTP extensions to provide parallel I/O support are outlined here. A design and implementation goal has been to provide for portability. Current ports exist on the IBM the RS 6000 supporting TCP/IP socket and IPI-3 over HIPPI data communication, the Intel Paragon supporting TCP/IP socket

data communication and Meiko supporting IPI-3 over HIPPI data communication. All implementations use TCP/IP control connections. The extensions also support partial file transfer, which we felt is going to be important as file sizes of 10s GB to terabytes become common. When we have had more experience with PFTP, we plan to turn the PFTP specification over to the Internet Engineering Task Force as a base for possible standardization. At the user command line level PFTP supports parallel append (Pappend), parallel get (Pget), and parallel put (Pput). During communication between the PFTP client code and the PFTP daemon on the HPSS side, the FTP port, store and retrieve commands have been extended to support the parallel file I/O and partial file transfer services. Because a seek is required for partial file transfers, parallel open and close commands have also been added.

We started with public domain standard FTP client and daemon code. The FTP client code had to be modified to support multiple Movers for handling the parallel data transfers and the extended commands. This required a little over 1000 lines of additional C source code to support both parallel TCP/IP and IPI-3 over HIPPI data transfers. The FTP daemon code had to be modified to support the extended commands and to build the IOD for communication with HPSS. This required about 100 additional lines of C source code.

Figure 9 illustrates the control and data flow for the parallel FTP interface to HPSS. This particular example is for a Pget operation. The numbers indicate flow sequence. The Parallel FTP Client determines the parallelism required at its end (a function of the stripe width of the local file, local I/O architectures and network connectivity), spawns Client Mover threads for each parallel connection which listen for connections, opens the local and remote files, passes the local client Mover ports to the daemon, issues a retrieve command, receives the parallel data streams, and issues reads/writes to the local storage (which might be a local parallel file system). The parallel FTP daemon builds an HPSS IOD and calls the HPSS Client API readlist function. This results in parallel data transfer controlled by the HPSS Movers.

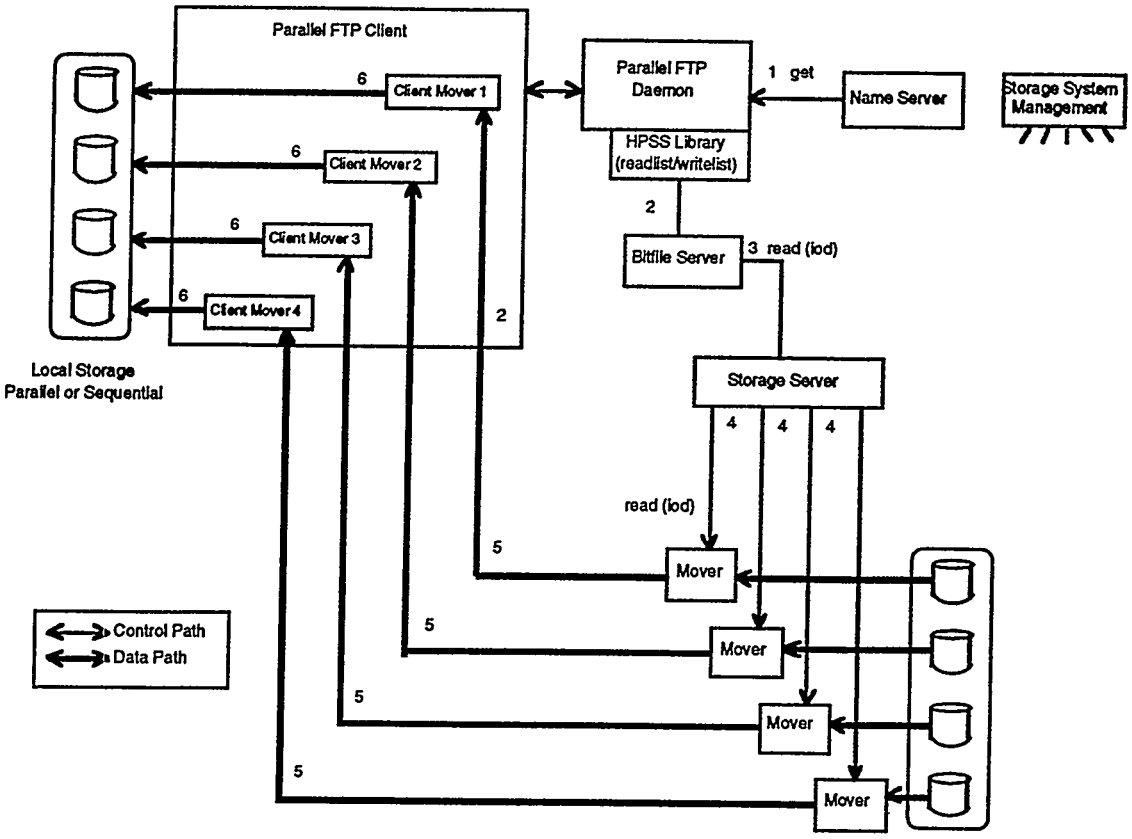


Figure 9. Parallel FTP transfer.

HPSS Integration with a Parallel System's Local Parallel File System

There is a spectrum of possible ways to integrate a parallel system's local parallel file system (PFS) and a storage system such as HPSS. The characteristic that defines where an integration exists on this spectrum is the transparency as seen by a user or application of the combined name space and data location between the local and storage file systems. At one end, which we will call the import/export end, an explicit command must be issued to move a file or partial file between a PFS and HPSS. At this end, two separate name spaces exist and multiple independent copies of the data exist. The use of FTP is an example of an integration mechanism at this end.

At the other end of the spectrum there is a single name space and data is automatically cached to the local PFS from HPSS as it is accessed using partial copies, or automatically migrates to HPSS when not recently accessed and space must be freed for the PFS. Thus, the application logically just sees a local PFS. which is a very large virtual store. The Unix Virtual File System (VFS) provides a model of a transparent integration mechanism for sequential

access and transfers. The Data Management Interface Group (DMIG) has specified mechanisms that use a local file system's name space as seen by applications but provides for transparent mechanisms to move whole files between local and remote systems. This approach does not, however, provide a uniform name space across multiple client systems. NFS and DFS server integrations with storage are examples of mechanisms to provide more transparency, but also do not support parallel I/O.

One would like to perform the integration such that high speed parallel transfers exist between applications and PFS, and between PFS and HPSS when appropriate connectivity supports parallel I/O. For example, NFS provides improvements in transparency but does not support parallel transfers to an application, although the HPSS NFS server does support high speed parallel transfers for caching and migration of large blocks. Similarly, the Andrew File System (AFS) [27] and Open Software Foundation's (OFS) Distributed File System (DFS).[30] support improvements in name space transparency over NFS, but again their protocols need to be modified to support parallel or even sequential high speed third-party transfers both to the application and between the a DFS server and HPSS.

The HPSS project is either using or planning to use existing industry standard mechanisms such as NFS, VFS, DMIG, and DFS to provide increasing transparent integration of HPSS and working with industry groups to improve the mechanisms and protocols to support transparent high speed parallel integration of HPSS with PFSs. An example is the Parallel Transport Protocol work and collaboration with the Scalable I/O Initiative [2]. There are, however, many issues needing resolution for full XXXXXXXXXXXXXXXXXXXXXXXXXX.

It should be noted that there are classes of applications and configurations where it may be most appropriate for applications to bypass access to either local PFS and directly to I/O to/from HPSS. Characteristics of such environments would be requirements for relatively large I/O transfers and an HPSS configuration with appropriate connectivity to the parallel system and number of resources.

We are also working with vendors and users to utilize the most effective import/export mechanisms available with a given PFS to integrate with HPSS. An example of this type of integration is the use of HPSS to support parallel tape services on the IBM SPx. In the initial integration, the tapes are connected to SPx nodes and the goal is to support high speed parallel import/export of data between the PFS controlled disks and the HPSS controlled tapes using the high speed internal SPx switching network.

HPSS to SPx PFS parallel-to-parallel transfer

Figure 10 illustrates the control and data flow for an HPSS to IBM SPx PFS parallel-to-parallel transfer. This particular example is for an export operation from PFS to HPSS. HPSS can run on a SPx node(s) or a separate server. The numbers indicate flow sequence. An HPSS import/export daemon is responsible for receiving the PFS requests, and communicating when the request has completed. For each request, an HPSS open API is issued, followed by a writelist API (for export), or readlist API (for import). The Mover has additional logic to issue PFS open and read (or write) APIs thus only a single Mover is required. The second Mover is implicitly in the PFS library and PFS. All other HPSS server processing is unaffected. Once the transfer has completed, the import/export daemon closes the HPSS file.

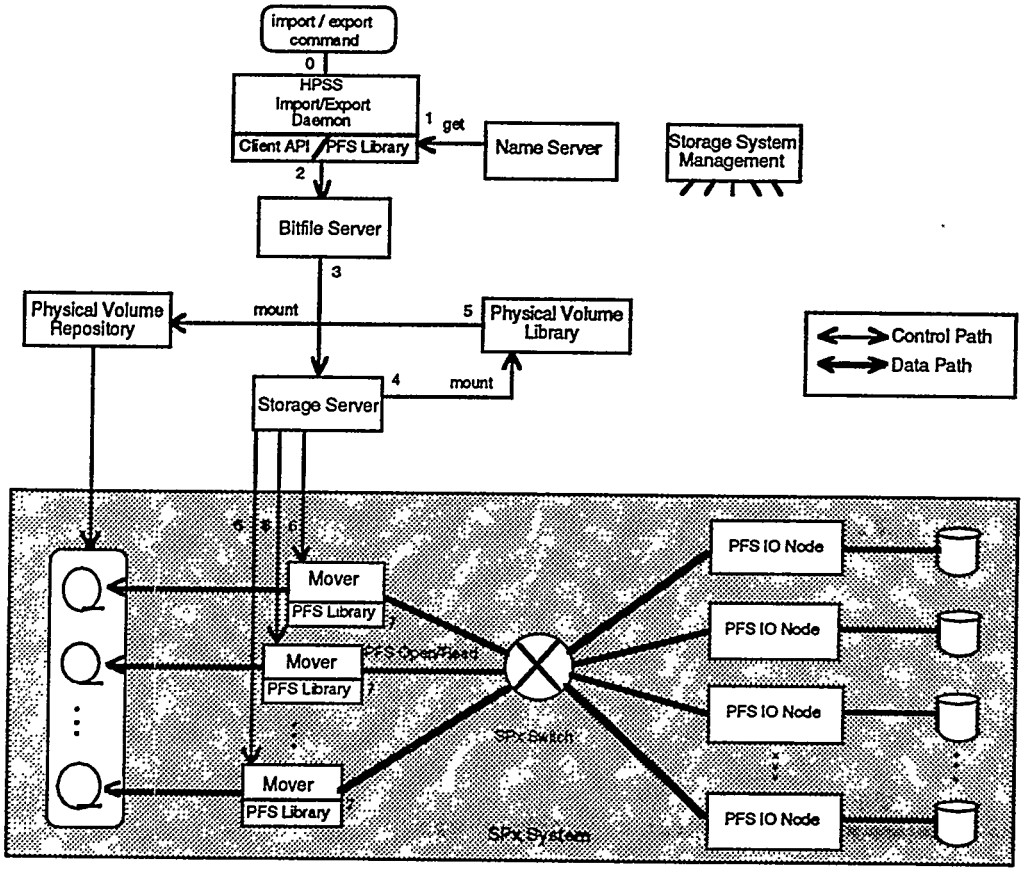


Figure 10. HPSS to SPx PFS parallel-to-parallel transfer.

(HPSS can run either on a SPx node(s) or separate server.)

Summary and Status

We have described the core parallel I/O mechanisms used in HPSS and its client interface services and given examples of their use. The key HPSS components involved in the parallel I/O architecture are the Storage Servers and the Movers. The central data structure used for describing a parallel transfer is the I/O descriptor (IOD) with its source and sink descriptor lists. The successive expansion of the IOD is it is passes through the system to create the parallel transfer plan and is a key architectural concept. The basic I/O transfer model is baaed on a general Parallel Transport Protocol (PTP).

Parallel I/O is a relatively new area of research, development and practical application [2,11,13,16]. There is much to be learned about parallel I/O and its use at application; library, language and compiler; operating system; local parallel file system; and parallel and distributed storage systems levels. As we learn more about the total end-to-end parallel I/O environment and how to integrate the various levels into easy to use parallel programming models, APIs, and tools we are confident that the modularity and parallel I/O architecture mechanisms of HPSS have the flexibility to evolve and integrate with the evolving total parallel I/O environment.

As this paper is being written, January 1995, HPSS Release 1 (R1) is in final integration testing and design is underway for integration with the Intel Paragon and Meiko parallel file systems. R1 supports the architecture described in this paper and supports parallel tape. Much of the coding for HPSS Release 2 (R2) has also been completed. R2 supports parallel disk, multiple hierarchies, and parallel migration and caching between storage devices at different levels of the hierarchy. General R2 availability is planned for late 1995.

HPSS parallel disk and tape I/O and parallel FTP were demonstrated at Supercomputing '94 in November 1994 in Washington, DC. The demonstration configuration in the HPSS booth shown in Figure 11 consisted of a four node IBM SP2, RS 6000 and PsiTech framebuffer with Sony monitor as client systems. Four SCSI disks on the SP2 nodes, 2 IBM 9570 Raid disks, 2 Ampex DST 600 D2 tape drives, 4 IBM 3480 tape drives and an IBM NTP drive were the available storage devices. The system was interconnected by a 32 node Network Systems Corp. HIPPI switch. Example data rates demonstrated, were 80 MB/s from the two 9570s to the SP2 (the limiting factor was the 40 MB/s SP2 channels), 26 MB/s from the two DST 600s to the framebuffer (we have also seen 39 MB/s from three

Ampex drives to an RS 6000 at the National Storage Laboratory) and 15 MB/s from the four SCSI drives to the framebuffer. Based on our observation of HPSS with our development and demonstration hardware we see no limits to scalability at this time introduced by the HPSS software. The limits observed are device and host channel rates, number of available devices and their connectivity and choices made for data layout such as stripe width and blocking factors at source and sink.

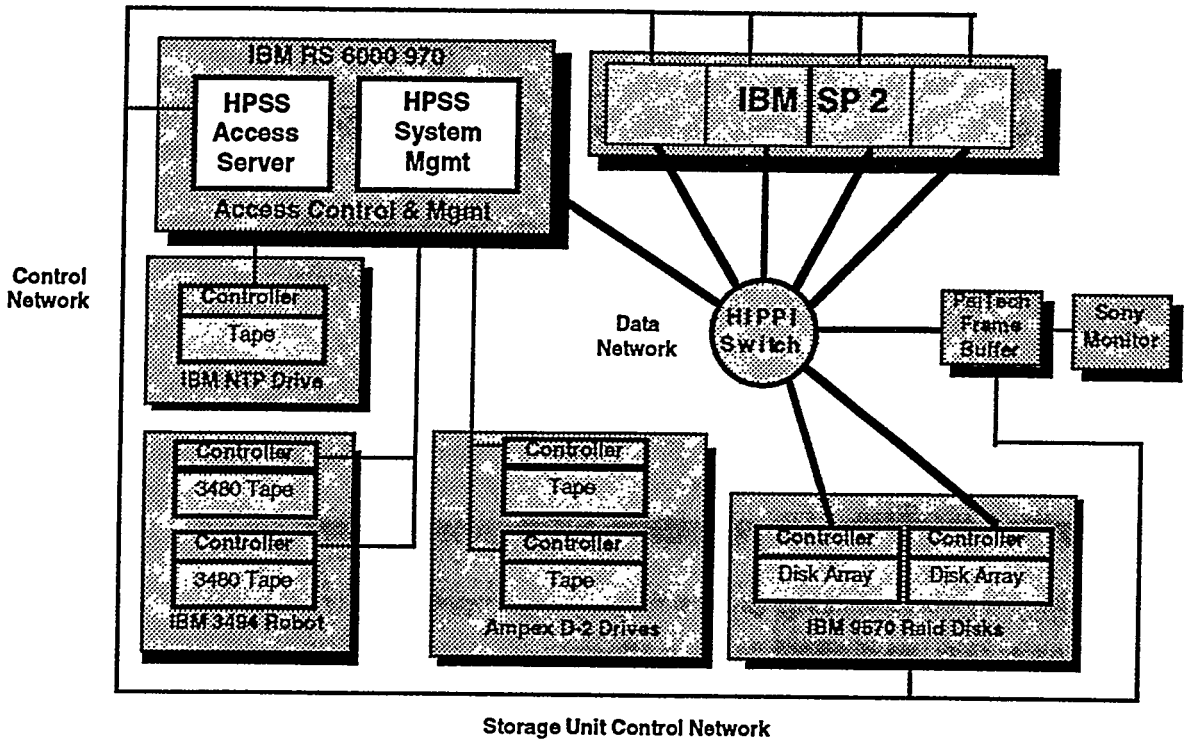


Figure 11. HPSS configuration used in the Supercomputing '94 demonstration.

(The RS 6000 was used as the HIPPI controller for the tape systems.)

Acknowledgement

The credit for the parallel I/O design and implementation work described in this paper belongs to the members of the Parallel Transport Protocol design group and the HPSS Technical Team: Specifically we want to acknowledge Peter Corbett, IBM TJ Watson, Kurt Everson, Rich Ruef, IBM, Erik DeBenedictis, Scalable Computing, Bill Nesheim, SUN Microsystem, Inc., and Larry Berdahl, LLNL for their work on the PTP design and Danny Cook of LANL, Dave Fisher of LLNL, Rena Haynes of SNL, and Kurt Everson and Rich Ruef of the IBM

team for their HPSS parallel I/O work. We wish to acknowledge the many discussions and shared design, implementation, and operation experiences with our colleagues in the National Storage Laboratory collaboration, the IEEE Mass Storage Systems and Technology Technical Committee, the IEEE Storage System Standards Working Group, and in the storage community. Specifically we wish to acknowledge the people on the HPSS Technical Committee and Development Teams. At the risk of leaving out a key colleague in this ever-growing collaboration, the authors wish to acknowledge Dwight Barrus, Ling-Ling Chen, Ron Christman, Danny Cook, Lynn Kluegel, Tyce McLarty, Christina Mercier, and Bart Parliman from LANL; Larry Berdahl, Jim Daveler, Dave Fisher, Mark Gary, Steve Louis, Donna Mecozzi, Jim Minton, and Norm Samuelson from LLNL; Marty Barnaby, Rena Haynes, Hilary Jones, Sue Kelly, and Bill Rahe from SNL; Randy Burris, Dan Million, Daryl Steinert, Vicky White, and John Wingenbach from ORNL; Donald Creig Humes, Juliet Pao, Travis Priest and Tim Starrin from NASA LaRC; Andy Hanushevsky, Lenny Silver, and Andrew Wyatt from Cornell; and Paul Chang, Jeff Deutsch, Kurt Everson, Rich Ruef, Tracy Tran, Terry Tyler, and Benny Wilbanks from IBM U.S. Federal and its contractors.

We also want to acknowledge that the edited description of the Parallel Transport Protocol was based on Reference [1] edited by Larry Berdahl and most of the detailed examples and figures were taken from an early HPSS description written by Danny Teaff of IBM.

This work was, in part, performed by the Lawrence Livermore National Laboratory, Los Alamos National Laboratory, Oak Ridge National Laboratory, and Sandia National Laboratories, under auspices of the U.S. Department of Energy Cooperative Research and Development Agreements, by Cornell, Lewis Research Center and Langley Research Center under auspices of the National Aeronautics and Space Agency and by IBM U.S. Federal under Independent Research and Development and other internal funding.

References

OLD NEW

1. 1. Berdahl, L., ed., "Parallel Transport Protocol," draft proposal, available from Lawrence Livermore National Laboratory, Dec. 1994.

2. 2. Bershad, B., et. al, "The Scalable I/O Initiative," White paper, available through the Concurrent Supercomputing Consortium, CalTech, Pasadena, Feb. 1993.

3. 3. Buck, A. L., and R. A. Coyne, Jr., "Dynamic Hierarchies and Optimization in Distributed Storage System," Digest of Papers, Eleventh IEEE Symposium on Mass Storage Systems, Oct. 7-10, 1991, IEEE Computer Society Press, pp. 85-91.

	4.	Carmen, T. H., and D. Katz, "Integrating Theory and Practice in Parallel File Systems," Proc. DAGS/PC Symposium 1993, pp. 64-74.
5.	5.	Christensen, G. S., W. R. Franta, and W. A. Petersen, "Future Directions of High-speed Networks for Distributed Storage Environments," Digest of Papers, Eleventh IEEE Symposium on Mass Storage Systems, Oct. 7-10, 1991, IEEE Computer Society Press, pp. 145-148.
6.	6.	Collins, B., et al., "Los Alamos HPDS: High-Speed Data Transfer," Proc. Twelfth IEEE Symposium on Mass Storage Systems, Monterey, April 1993.
	7.	Corbett, P. et al., "MPI-IO: a Parallel File I/OP Interface for MPI, version 3, available at http://lovelace.nas.nasa.gov/MPI-IO/mpi-io.html,"
	8.	Corbett, P. F., and D. G. Tectelson, "Design and Implementation of the Vesta Parallel File System," Proc. Scalable High Performance Computing Conference, 1994, pp. 63-70.
7.	9.	Corbett, P. F., D. G. Feitelson, S. J. Baylor, and J. Prost, "Parallel Access to Files in the Vesta File System," Proceeding of Supercomputing '93, IEEE Computer Society Press, Nov. 1993.
	10.	Corbett, P. F., D. G. Feitelson, S. J. Baylor, J. Prost, "Parallel Access to Files in the Vesta File System," Proceeding of Supercomputing '93, IEEE Computer Society Press, November, 1993.
8.	11.	Coyne, R. A. and H. Hulen, "An Introduction to the Mass Storage System Reference Model, Version 5," Proc. Twelfth IEEE Symposium on Mass Storage Systems, Monterey, April 1993.
9.	12.	Coyne, R. A., H. Hulen, and R. W. Watson, "Storage Systems for National Information Assets," Proc. Supercomputing 92, Minneapolis, Nov. 1992, pp. 626-633.
10.	13.	Coyne, R. A., H. Hulen, and R. W. Watson, "The High Performance Storage System," Proc. Supercomputing 93, Portland, IEEE Computer Society Press, Nov. 1993.
11.	14.	DeBenedictus, E., and S. Johnson, "Extending Unix for Scalable Computing," to appear in IEEE Computer, Nov. 1993.
	15.	del Rasario, J. M. and A. Choudhary, "High Performance I/O for Parallel Computers: Problems and Prospects" IEEE Computer, 27 (3): 59-68, March 1994.
12.	16.	Deutsch, J., and N. Samuelson, in the 10s GB to terabytes range IEEE Symposium paper
13.	17.	Dibble, P. C., "A Parallel Interleaved File System," Ph.D. Thesis, Univ. of Rochester, 1989.
14.	18.	Dietzen, Scott, Transarc Corporation, "Distributed Transaction Processing with Encina and the OSF/DCE," Sept. 1992, 22 pages.
15.	19.	**Fisher, Dave, IEEE Symposium paper**
16.	20.	Ghosh, J. and B. Agarwal, "Parallel I/O Subsystems for Distributed Memory Multicomputers," Proceedings of the Fifth International Parallel Processing Symposium, May 1991.
17.	21.	Golubchik, L., R. R. Muntz, and R. W. Watson. "Analysis of Striping Techniques in Robotic Storage Libraries." Submitted to the 14th IEEE Symposium on Mass Storage Systems, Monterey, Sept. 1995. Appeared as a poster at Supercomputing '94, Washington, D.C., Nov. 1994.
	22.	Hartman, J. H., and J. K., Ousterhaut, "The Zebra Striped Network File," Proc. 14th ACM Symposium on Operating Systems Principles 1993, pp. 29-43.
18.	23.	Hyer, R., R. Ruef, and R. W. Watson, "High Performance Direct Network Data Transfers at the National Storage Laboratory," Proceedings of the Twelfth IEEE Symposium on Mass Storage, Monterey, IEEE Computer Society Press, April 1993.
19.	24.	IEEE Storage System Standards Working Group (SSSWG) (Project 1244), "Reference Model for Open Storage Systems Interconnection, Mass Storage Reference Model Version 5," Sept. 1994. Available from the IEEE SSSWG Technical Editor Richard Garrison, Martin Marietta (215) 532-6746
20.	25.	IEEE Storage System Standards Working Group, "Draft Mover Specification," in preparation.
21.	26.	"Information Technology - Open Systems Interconnection - Structure of Management Information - Part 4: Guidelines for the Definition of Management Objects," ISO/IEC 10165-4, 1991.
22.	27.	Internet Standards. The official Internet standards are defined by RFC's (TCP protocol suite). RFC 783; TCP standard defined. RFC 959; FTP protocol standard. RFC 1068; FTP use in third-party transfers. RFC 1094; NFS standard defined. RFC 1057; RPC standard defined.

23.	28.	ISO/IEC DIS 10040 Information Processing Systems - Open Systems Interconnection - Systems Management Overview, 1991.
24.	29.	Katz, R. H., "High Performance Network and Channel-Based Storage," *Proceedings of the IEEE*, Vol. 80, No. 8, pp. 1238-1262, August 1992.
25.	30.	Lampson, B. W., M. Paul, and H. J. Siegert (eds.), "Distributed Systems - Architecture and Implementation," Berlin and New York: Springer-Verlag, 1981.
4.	31.	Louis, S., and R. Burris, "Management Issues for High Performance Storage Systems," Draft Submission Fourteenth IEEE Computer Society Mass Storage Systems Symposium, Sept. 11-14, 1995
26.	32.	Louis, S., "Class of Service in the High Performance Storage System," IFIP International Conference on Open Distributed Processing Brisband, Australia, Feb. 1995
27.	33.	Morris, J. H., et al., "Andrew: A Distributed Personal Computing Environment," Comm. of the ACM, Vol. 29, No. 3, March 1986.
28.	34.	Nelson, M., et al., "The National Center for Atmospheric Research Mass Storage System," Digest of Papers, Eighth IEEE Symposium on Mass Storage Systems, May 1987, pp. 12-20.
	35.	Nydict, D. et al., "An AFS-based Mass Storage System at the Pittsburgh Supercomputer Center" Digest of Papers Eleventh IEEE Symposium on Mass Storage Systems, Oct., 1991, pp. 117-122.
29.	36.	Open Software Foundation, Distributed Computing Environment Version 1.0 Documentation Set. Open Software Foundation, Cambridge, Mass. 1992.
30.	37.	OSF, File Systems in a Distributed Computing Environment, White Paper, Open Software Foundation, Cambridge, MA, July 1991.
	38.	Peterson, D. G., and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," Proc. SIGMOD Int. Conf. on Data Management, Chicago 1988, pp. 109-116.
	39.	Rellman, B., and D. Payne, "An Efficient File I/O Interface for Parallel Applications," Draft Working Paper prepared for the Scalable I/O Workshop Frontiers '95, had @ssd.intel.com
31.	40.	Sandberg, R., et al., "Design and Implementation of the SUN Network File System," Proc. USENIX Summer Conf., June 1989, pp. 119-130.
33.	41.	Teaff, D., R. W. Watson, and R. A. Coyne, "The Architecture of the High Performance Storage System (HPSS)," Proceedings of the Goddard Conference on Mass Storage & Technologies, College Park, March 1995
32.	**42.**	**Teaff, Danny, Check to get permission (marked confidential) to ref PFTP spec**
34.	43.	Tolmie, D. E., "Local Area Gigabit Networking," Digest of Papers, Eleventh IEEE Symposium on Mass Storage Systems, Oct. 7-10, 1991, IEEE Computer Society Press, pp. 11-16.
35.	44.	Watson, R. W., R. A. Coyne, "The National Storage Laboratory: Overview and Status," Proc. Thirteenth IEEE Symposium on Mass Storage Systems, Annecy France, June 12-15, 1994, pp. 39-43.
36.	45.	Witte, L. D., "Computer Networks and Distributed Systems," IEEE Computer, Vol. 24, No. 9, Sept. 1991, pp. 67-77.

Appendix A. Detailed outline of the parallel transport protocol

Initially, all servers are listening on their well-known public control ports. The Client receives a user or application request (e.g., Transfer (Source GatherList, Sink ScatterList) and starts the transfer. The transfer may involve two or three parties. The Client may be associated with either of the servers or be a third party. In the two-party case, the Client and the passive server are the same party; the functions described below for the passive server

are then performed internally within the collocated Client and Server. The two-party case is expected to be the most common case.

Client to passive server

The client

The Client generates a unique transfer identifier for this transfer which will be used in all messages exchanged among the parties involved in the transfer (e.g., this will be used to identify the group of parallel transactions as parts of a single logical transfer). The Client chooses one server to be passive and the other to be active. The Client connects to the passive server using the well-known network control address of this server, and sends it an ***IOD*** message. The ***IOD*** contains the command to move data (read or write) relevant for that server, and the appropriate source or sink-descriptor lists. The message also requests a set of network ports mapped to window offsets and lengths. If the passive server is the data source, then the gather list is sent, otherwise, the scatter list is sent.

The client waits for one or more replies (***IOR*** messages) from the set of servers associated with the passive side of the transfer (replies to the client's request may come from many distributed servers involved in a logical server). When all of the expanded IOD lists requested have been accounted for in the IORs returned to the Client, the Client can proceed.

The passive server

The passive server (listening for client connects):

- receives the *IOD* describing the data to be transferred,
- determines the physical media involved in storing and moving the data,
- determines device direct transfer capability,
- generates and associates control ports with the movement of data such that local move rates can be maximized,
- maps device,data offset, lengths to the data ports, and window offsets and length to be used for transmission,
- determines data port protocol options, and

- replies to the Client with this information in an *IOR* message. (With this information, active server ports will know the addresses of the passive server ports to connect to in order to setup the data port data exchange and knows how the passive server's data is mapped to or form the window.) Listens are posted on these ports to wait for connect requests (possibly many ports are listening for possibly many simultaneous connections). Data transmission logically begins when the connect requests are received (connect and transfer, below).

In the special case where no additional control exchanges would be necessary to setup the actual data transfer over the data connections, data port, rather than control port, addresses are returned in the *IOR* to the client, and the data ports post listens. This will be a frequent case.

Client to active server.

The client

The Client constructs an *IOD* for the active server, when all IORs have been returned from the passive server, which contains:

- the relevant command to move data (read or write) and an IOD containing the passive server's descriptor list describing (1) the control port addresses, data port protocol options, and the transfer assignments (window mappings) from all of the *IOR*s received from the passive servers, and (2) the logical transfer assignments (a scatter (write) or gather (read) list as appropriate for the active server).

The Client connects to the active Server's well-known public control port, and sends the IOD.

The active server

The Active Server receives the *IOD* describing the data move, examines the part of the IOD structure describing its half of the transfer (the scatter(write) or gather(read) distribution list, whichever is relevant, and

- determines the physical media involved,
- determines direct device transfer capabilities of the media,
- generates control ports to maximize local move rates, if needed,

- maps the transfer window assignments implied in the IOD to the control ports (assigns the control ports a portion of the transfer window),
- determines the port to port interconnect between the servers (data transport plan) by reading the passive server's control port list and transfer data window assignments supplied in the **IOD** constructed by the Client from the passive server(s) **IORs**, and
- determines the data port protocol to be used on a connection basis.

Connect control and data transfer between passive and active servers

All of the following connect and transfer activities occur in parallel. In the special case where data ports were passed to the active server, only data port connect and transfer below is performed (in this case it was found that no additional control exchanges would be necessary to setup and control the data transfer over the data connections and therefore data port addresses were supplied by the Client). All passive server ports assigned to this specific transfer are currently listening for active port connects.

Active control ports

Active control ports establish connections with each of their (passive) peers. Multiple connections will be established in parallel if multiple control ports are involved on the active side. Further, a given port on either side may be involved in several connections, depending on the required data transfer plan.

Determine connection control

Data ports are created or reused as necessary. An Initiator message is sent to the passive server that contains

- the data port protocol to use,
- the transfer window data identification (window offset and length),

- which side will assume control of the data connection (this depends for example on the device type at each end), and
- if the passive side data port will control the data connection (a role reversal), the data port address it is to connect to (the active side data ports post a listens in this case before the message is sent).

Passive control ports

Passive control ports reply with an Initiator message containing the data port address to connect to, and the window offset and length of the data to be moved. The amount of data moved between the data ports will be the minimum of the lengths exchanged between the two control ports (buffers may mismatch).

Data port connect and data transfer:

Data ports connect. Active ports connect to passive ports in parallel. There may be multiple connections if multiple data ports are involved on either side.

Transfer Data. Data is transferred in parallel for each independent data port connection pair. Data to be transferred is identified by a data tag {transaction identifier, offset, size}. The active port will either *pull* or *push* the data in a desired order. When *pushing*, the data tag is sent to the passive port followed by the data . When *pulling* , the data tag is sent to the passive port requesting the data, and the passive port responds with the data tag followed by the data. This action is not necessary if the underlying protocol already has an equivalent mechanism. For an example of the type of protocol used on a given connection see Reference [18].

Completion status. Completion status is returned to the control port.

Iterate until all data transferred. Active control ports may optionally send a completion message to the respective passive control port giving the completion status and bytes moved before iterating.

Termination

Each server collects completion status from the ports assigned to the transfer, forms a completion status list corresponding to the assigned transfer responsibilities, and reports to the Client with an IOR completion message. The Client (listening for server replies) determines IO completion when the server(s) report [only one server can report with an IOR in a two party transfer]. Server control ports are closed. Completion status is returned to the user. The transfer terminates.