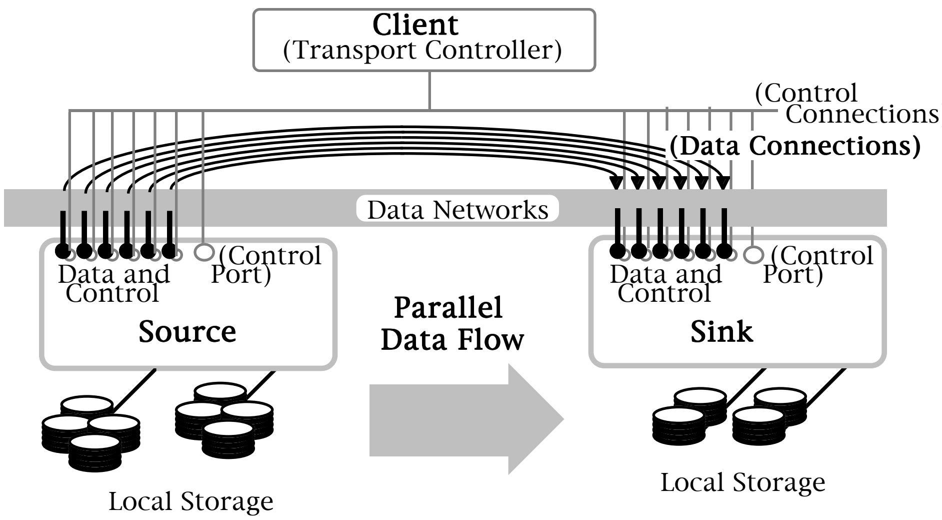
Parallel Transport Protocol Proposal

DRAFT

January 3, 1995



Data is transferred in parallel using multiple control and data connections during a single transfer. The serial external interface bottleneck between systems is eliminated. Bandwidth scales transparently with increased physical parallelism. Existing transport protocols are used.

This proposal results from a consensus industry decision to pursue a data transport protocol that would be appropriate to address increasing data distribution arising from client/server computing and increasing bandwidth requirements arising from massively parallel computing. Parallel transport does not exclude or impeed sequential technology alternatives from developing at their normal pace.

Introduction

This document defines a Parallel Transport Protocol (PTP) that permits parallel data flow between a wide variety of interconnected equipment configurations. The intent of the protocol is to provide bandwidth improvements for large transfers which scale implicitly with increasing physical parallelism (e.g. with increasing numbers of devices or increasing network parallel interconnectivity). Parallel transport will be defined as a connection mode transport application which will use standard transport providers for data movement, and standard transport interfaces for implementation.

The protocol will define:

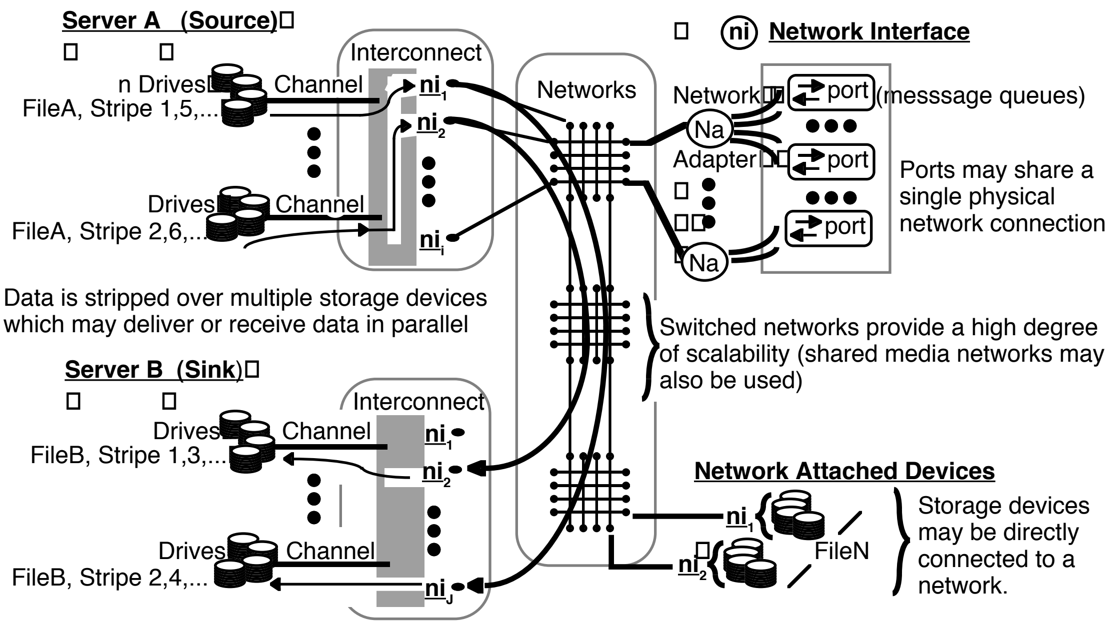
1. A procedure for managing parallel data flow.
2. The required data structures to be exchanged.

Data will be moved between sources and sinks in parallel. Any combination of segments of files, physical devices, memory buffers, or network locations may be the source or sink of the data.

Design Objectives

1. Define a parallel external interface between heterogeneous systems: provide a parallel transport definition between sources and sinks.
2. Provide for parallel transport of very large data sets. Movement of large data sets is the prime motivation for the development of this protocol.
3. Provide IO bandwidth improvements to applications which scale implicitly with increasing physical parallel connectivity for large data transfers.
4. Maintain architectural independence. The protocol must be suitable for use in existing sequential IO environments. Any combination of parallel and sequential architecture's may be the source or destination of data.
5. Maintain transport protocol independence. Maintain maximum interoperability among networks.
6. Support network attached peripherals.
7. Support gather/scatter and random access. (Establish a general data selection methodology amenable for application programming interfaces and employable between a variety of systems.)

This document does not define an API; it defines the underlying parallel transport machinery to be used by existing or future APIs. Parallelism expressed by these APIs may be either explicit or implicit depending on the API. In particular, it is not necessary to change the application to realize scalable IO improvements. Parallelism is independent of the underlying physical transports. Implementations of the protocol can easily adapt to increasing device or network parallelism.



Two machines transferring data in parallel over a switched network

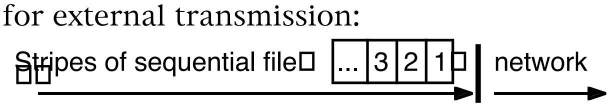
Data to be transferred is stripped over multiple storage devices or buffers within the host machine interconnects. Machine interconnects, regardless of topology, serve to route data to or from **one** or more **Network Interfaces.** Logical control and data ports may share one or more physical network adapters. High performance switched networks are most appropriate for the protocol. Above, two network interfaces are being used for a parallel transfer over a switched network. A parallel transfer involves movement of data from one or more sending (source) devices to one or more receiving (sink) devices over **one** or more network interfaces. An established connection represents a complete end-to-end data path: data moves from the source devices, over the channels to the sending host, through the host Interrconnect, to the ports where it is moved to a network adapter (Na) for transmission. The reverse path is followed on the sink side. Networks are by definition unpredictable transmission mediums: it is not possible to say when portions of a transfer begin or what the rates are on a given logical connection.

Fig. 2 Physical Connectivity and Data Flow.

Background

It has been common practice for many years to store data in a manner that facilitates parallel access. Multiple storage devices are used to increase host to device bandwidth. A common technique is to store data in *stripe*s across multiple independent storage devices. Above, for example, *File* has been divided into blocks and allocated to multiple devices which are on separate *channels* (*File* has been *stripped* across devices and channels). A reference that involves physically distributed data, on multiple devices which operate in parallel and without conflict, has a total potential bandwidth with the host which is the sum of the bandwidths of the storage devices connected to the channels (or the sum of the *channel* capacities).

In contrast, data must be ordered and sequential in existing parallel systems for transport over external interfaces. In Fig. 2, *FileA* would be placed in a sequential stream for external transmission:



The parallelism available locally is thus lost at the network interface; the network, and the interfaces on both ends of the transmission, remain a sequential data path and a potential bottleneck. There is currently no available alternative to this sequential ordering requirement for external transmission. Although the data is stored in parallel locally, it is highly dependent on the architecture of the local storage configuration; there is no accepted convention for relating the local parallelism between hosts. A single *logical* connection, which forces a sequential transfer, is currently required for data transport between external machine interfaces.

Goal The goal of Parallel Transport is to enable bandwidth improvements which scale with increasing physical parallelism. Bandwidth will improve with the ability to transport data in parallel that is stored in parallel. The protocol enables parallelism which is available locally to be expressed across external machine interfaces between machines. A closely related goal is to move data in parallel which is distributed among multiple servers (across network locations) and logically related (e.g. part of the same file). In Fig. 2, *FileN* is physically distributed over independent network attached storage devices. The protocol would enable the data to be moved, in parallel, in a single request.

Method The protocol uses multiple *logical* connections for a *single* transfer and proposes a method for relating the data transferred among these connections as parts of a single logical transfer. Logical connections are bound to available physical transports using existing transport interfaces without change. Scalable speedup occurs dependent on how the connections are used i.e. dependent on whether they represent independent data paths, whether bottlenecks in the data flow can be parallelized as a result of their use and dependent on the local and external physical configuration.

Application Multiple connections for a single transfer can be used to potentially maximize the rate of flow between local devices and external interfaces. If both sides of a transfer conduct a policy which potentially maximizes local transfer rates, the overall transfer rate can potentially be maximized between them. For example, a connection can be assigned, locally, to each independent data path involved in a transfer. In Fig. 2, we choose to assign one connection per channel. The analogy, in this case, would be to stripe the data to be transferred across the external connections as it is stripped across the channels. A different strategy could have been chosen. The protocol does not need to establish what the local policy should be for connection use. This will be implementation specific and architecturally dependent. An important constraint, is that the cost of connection management remain small relative to the data transfer: the amount of data per connection cannot become arbitrarily small.

Imbalances and Path Conflicts If no constraint is placed on the number of connections, it is possible that large imbalances could occur (e.g. 1000 ports on one side establishing connections with 2 on the other). In practice a policy of maximizing local transfer rates, subject to minimizing the cost of connection management relative to the size of the data transfer, should sufficiently limit the number of connections. An implementation can use a network transmission rate approximation for a specific network as a guide and limit if necessary on the number of connections. If for example, a 100MB/s network is to be used for possibly many 10MB/s local devices, then it may be beneficial to use 10, rather than 20 connections for the transfer.

Overall Data Flow A data transfer will be viewed as multiple complete end-to-end paths: from devices of the sender to devices of the receiver (a buffer within the machine interconnect is viewed as another device variant). Internal machine interconnects are highly architecturally specific; from the functional view of moving data, all interconnects serve as a switch to route the data between the external interfaces and the local storage devices.

Data flows from the devices of the sender, through the channels, through the host interconnect, possibly to specialized IO nodes, through the network interface, where it is moved to the network for transmission. On the receiving side, data moves from the network to the receiver's network adapter, through its network interface, over the interconnect, to possibly one or more memory buffers within the interconnect, or over the channels to the devices of the receiver. A connection represents one such data path. A connection sharing either a source or destination with another connection may introduce path conflicts which may degrade the flow rate for these connections.

Rates of Data Flow The maximum rate that data can be moved over each connection is the minimum rate of any of the dependent processing components in a path i.e. one component will limit the move rate on a connection. This limit could be presented by the physical transport media, the network adapter, protocol processing, repeated memory buffering, the interconnect, slow devices or other processing components involved in the movement of the data.

	Devices		Machine	Network	Network
	Positioning	Transfer	Interconnect	Interface	Transmission
Rate	5Ms	5MB/s	200MB/s	1000MB/s	100MB/s
Relative to Device		1	40	200	20

(Rate/5MBs, ignoring positioning times)

Fig. 2a Example system data movement rates

In the above configuration, for example, a transfer would be limited first by the positioning times of the devices involved, followed by the device transfer rate. If the data is stripped over 20 or more devices, it is possible to reach maximum network transmission speeds of a single interconnect which would then present the next limit. Finally, if multiple network connections are available, it is possible to reach machine interconnect rates.

There are many potential bottlenecks to the data flow along a connection. These will vary between architectures, system, and networking interface implementations, and physical transport technology. The use of multiple connections, as proposed here, parallelizes the entire flow path and therefore, offers the greatest potential for removing bottlenecks regardless of the system architecture, how the network interfaces have been implemented, or what the physical transport medium is.

Rate Matching A parallel transfer is a set of independent transfers which compete for network bandwidth along with all other transfers occurring simultaneously (both sending and receiving). Flow control is assumed on a per connection basis. It is neither necessary nor practical to attempt to match the overall sending rate with the overall receiving rate for a single transfer. The maximum rate any set of transfers can occur is the minimum of the maximum sending rate, transport rate, and receiving rate (all of which may vary constantly). Because either side, or the network, present constantly varying transfer rates per connection, it is not possible to guarantee a given overall transfer rate.

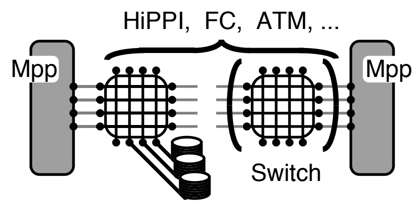


Fig. 3 Deployment in high performance switched LANs

Host Networks The protocol is not directly dependent on a particular underlying physical transport and is appropriate for general networking environments. The environments where this protocol is expected to be most commonly used are switched local or wide area networks, where multiple physical network connections per machine are possible and multiple physically parallel paths among hosts and devices in the local area are possible. (Shared media networks may quickly saturate with increasing device parallelism.)

The manner in which logical connections are mapped to multiple physical transport interfaces within a particular system and network environment is architecturally and implementationally specific to that environment and cannot be specified by the protocol (the logical to physical port mapping is established by the transport layer interface used, is implementation dependent, and is thus not specified by the protocol). Whether or not multiple physical network connections are used during a transfer is load and configuration dependent.

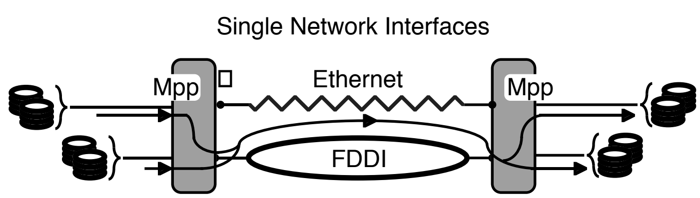


Fig. 4 Deployment in low performance shared media LANs

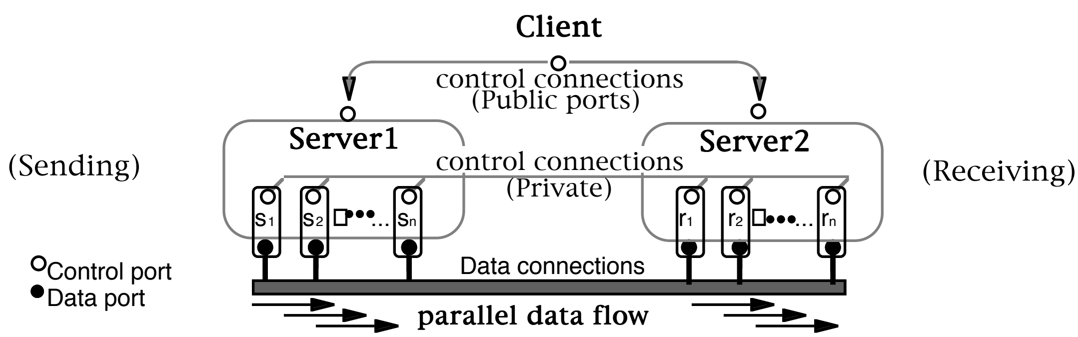
Physically separate network transport paths and multiple network connections may not be necessary to achieve useful speedup. (Suppose a single infinite speed link were available and shared by all connections: the transport medium would not present a limit.) In most cases, multiple logical ports can be mapped to a single network adapter and there will still be speedup, up to the limit of the network connection involved. In example configuration Fig 2a, it is possible to reach a 20 fold improvement using a single network connection. Even if the transport medium were to have been a single shared media sequential path (e.g. a 100Mbs FDDI ring), a useful speed up (of 2) would have been achieved.

__Mixing Transport Media and Protocols__ It is expected that control and data connections **will** use a different transport media if these are available. For example, TCP/IP could be used for control and IPI-3 for data. However, protocol mixture during a single transfer is not expected to be beneficial or desirable for either control or data connections. In most cases, a single protocol will be used for data connections and a single protocol will be used for control connections during a given transfer. The protocol chosen for data connections will apply for all the data connections throughout the transfer (and similarly for control connections). For example, data connection protocols SCSI-3 and IPI-3 would not be mixed during a single transfer: the protocol would be chosen in advance and apply to the entire transfer.

The proposal provides for protocol mixture, within a transfer, on a limited basis only (i.e. IP and IPI-3 for data connections). The networks must be known in advance to be fully interconnected when multiple protocols are used (every node on one network must reach every node on the other; this is not guaranteed over multiple protocols).

Intelligent peripherals directly attached to a network are not currently common on most networks. The protocol anticipates an increasingly important role for network attached devices, and makes specific provision for the capabilities of intelligent peripherals in the protocol. See **Definitions and Terminology** for IPI-3 protocol definitions, and its use in **Example 2 part 2**.

__Applicable Area__ While the protocol is to be most commonly used on high performance networks, there are no design restrictions introduced which would prohibit its use over a wide area. Geographical separation of client and servers is not logically relevant for either control or data connections.



Public control ports are used to initiate and setup a parallel transfer. Private control ports are created and associated with one or more data ports when additional connection control is necessary for data connections during a transfer. Private control and data ports are created specific to and exist during a given transfer only. All private connections are established, and operate, in parallel. Data is transferred between a set of sending {s1,2..} and a set of receiving {r1,2...} data ports. A full logical interconnect between private control ports, and between the data ports, may be necessary for a parallel transfer. The client may be either of two servers or a third party (as shown). (There are no architectural destinctions among clients and servers; these may be MPPs, workstation clusters, personal computers, intelligent peripherals or any other.)

Fig. 5 Parallel Transport Model

Parallel Transport Model

Connection Mode Parallel transport will be defined for connection mode transport services. Connection mode transport services define a method for two conversing entities to establish a logical connection, transfer data, and close the connection. Establishing a connection is naturally an asymmetric process: one side most post listens (the passive side) and the other must actively connect and control the connection (the active side).

Connection Classes Two classes of connections will be used: *control* and *data*. Control and data connections are logically separate and may or may not share the same physical data path. There will be an advantage to use lower latency networks for small packets characterized by control exchanges and high bandwidth networks for large data transfers. *Control connections* will be used for commands and replies to commands. *Data connections* will be used for data movement.

Clients and Servers A *client* initiates requests for service on *control connections*; *servers* provide the data transfer service on *data connections*. Clients and servers may be MPPs, workstation clusters, a single machine, or intelligent peripherals: there are no architectural distinctions. A *client* is defined by initiating a request for service, and may thereafter act as a server by performing parts of the request.

Two and Three Party Transfer A transfer may initially involve either two or three parties; a client may be one of two servers or a third party. A three party transfer allows a client to initiate and control a transfer between a pair of servers where the client is neither the source or destination of data (it is a third party). For a two party transfer (the predominant case), the client will be one of the servers and the actions described for that server will apply to the client in this case. A server may in turn contact other servers to perform the client's request (and would therefore be acting in a third party client role).

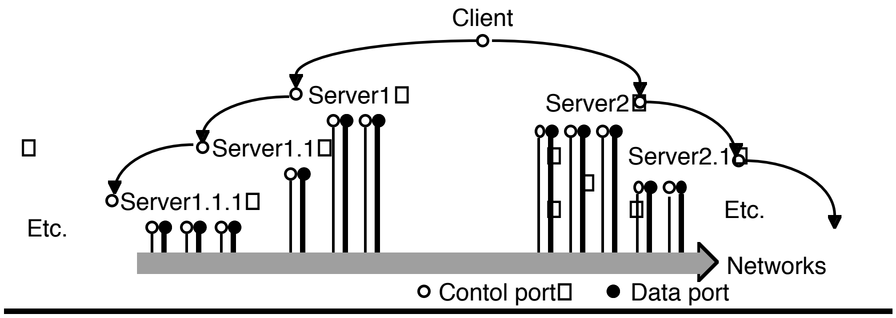
Public and Private Connections Well known network *control address*es (of the *control ports*) will be publicly available for the client and servers (known to all hosts on the network). Public control connections are used to initiate, control, and describe the transfer; these are setup exchanges which are sequential. Private control and data connections are created specific to the transfer. Private control connections may be established during the transfer to control specific data connections. Data is moved in parallel over sets of data connections between storage devices in the sending host to storage devices in the receiving host. Data connections will be private (i.e. created specific to a transfer).

Use of Connections A set of *data* connections will be used to move data in parallel (in contrast to the single logical connection which is now required) and a set of private control connections will be used to control the data connections (in parallel). At the lowest level, parallel transport is a simple set to set transfer among sets of sending and receiving data ports. The model seeks to introduce parallelism into the transport mechanics as the means to increase bandwidth in a manner which scales implicitly with, and utilizes, increasing physical transport parallelism. *Logical* connections are mapped onto any underlying physical transport media by the transport layer interface used for implementation.

The mechanics underlying parallel transport are as follows:

1. The Client connects to the well known control port of the passive server and sends a command to move data. One or more servers reply to this request with a set of ports (network addresses) to use for the transfer. These are either control ports to be used to establish and control data connections, or the data port addresses to use for the transfer. All ports listen for connect requests from active servers.
2. The Client connects to the well known control port of the active server, sends the list of ports assigned to the transfer by the passive server, and the relevant command to move data (i.e. a read or write command with a description of the data location).
3. Private control ports connect (in parallel), and exchange information necessary to set up the transfer on a per port basis; data ports connect and transfer data in parallel. All connections operate logically in parallel.
4. The Servers reply to the Client with completion status.

Distribution of Data An application specifies the logical data to be moved; physical device allocation will not typically be known. Data to be transferred may require random access, implicitly specify physical devices, or be distributed over multiple servers and remote devices. A variety of device types may be involved when moving data. The devices may be intelligent peripherals directly attached to the network.



In general, the data to be transferred may be distributed over sets of devices and servers anywhere on the network. Each server may be required to contact other servers to perform parts of the transfer. A parallel transfer may therefore involve a general network wide 'gather' operation, and a network wide 'scatter' operation.

<u>Connectivity Requirements</u> Each *Server* will manage the movement of data between its physical devices and data ports. The protocol will not introduce dependencies on how data is distributed within the server (this will be architecturally specific to both the data management system, and the hardware involved). It is not practical to impose a data ordering on an application.

When sets of ports are used for a transfer where no ordering of the data can be assumed, it may be necessary for each sending port to deliver a part of the transfer to each receiving port. In contrast to a single pair of ports, the general data delivery requirement for parallel transport becomes the cross product of the ports involved. This imposes **full interconnectivity** requirements on network connections.

> If s_i is a set of sending *ports* $\{s_1,s_2,...\}$, and r_i is a set of receiving *port*s $\{r_1,sr_2,...\}$, and no initial ordering is imposed on the transfer then
>
> $$\{s_1,s_2,...\} \dashrightarrow \{r_1,sr_2,...\}$$
>
> may require the cross product of the subsystems
>
> $s_1 \rightarrow r_1$, $s_1 \rightarrow r_2$, $s_2 \rightarrow r_1$, $s_2 \rightarrow r_2$, ...
>
> where each *port* sending data may be required to deliver a portion of the data to each *port* receiving data to accomplish the transfer.

Multiple connections between the private control and data port addresses on each side must be possible: a full logical interconnect must be possible. Networks must serve as a logical fully connected switch to achieve a cross product of the ports on each side for parallel transport to be realized.

Parallel Transport Protocol

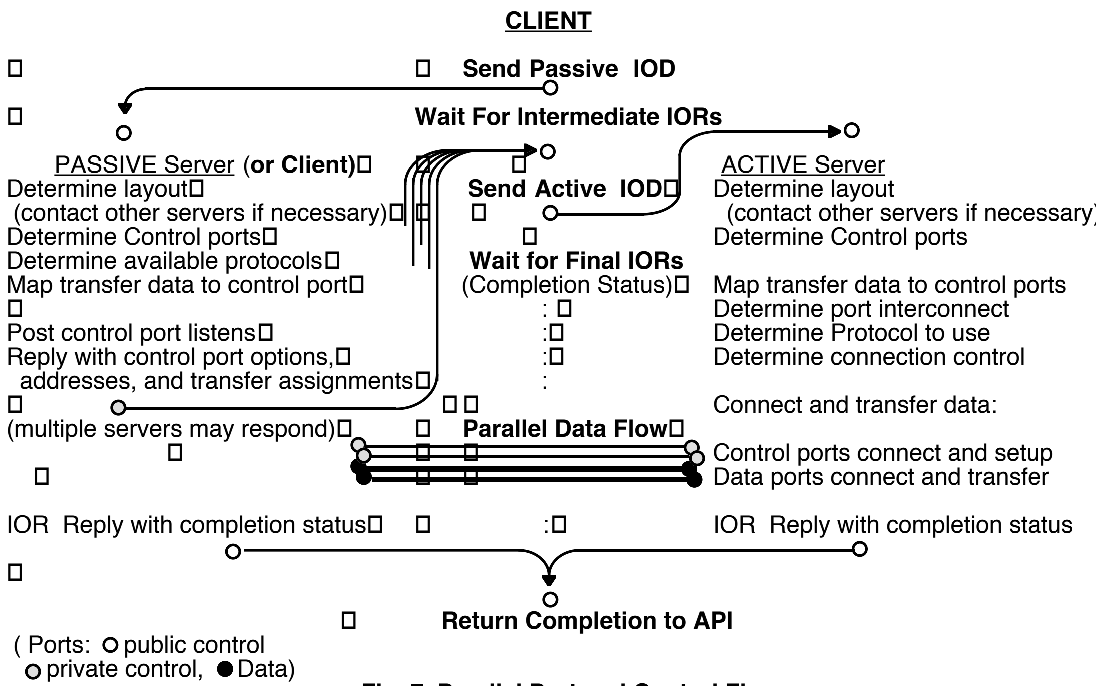


Fig. 7 Parallel Protocol Control Flow

<u>Application Interface</u> An application running on the Client will make an IO request. The request designates a source server with a gather list that specifies the data to be sent, and a sink server with a scatter list that specifies where data is to be sent. The source and sink servers are designated by their public control addresses on the network. For example, an interface to parallel transport might appear as follows (client APIs are not part of the protocol):

Transfer(Source,GatherList, Sink,ScatterList)

<u>Active and Passive Roles</u> The client (i.e. an implementation of the protocol on the client) receives the user request and begins the transfer. The client will assign an *active* role to either of the servers and a passive role to the other. The connection role chosen for a server is not dependent on whether the server is the source or the sink of the transfer. Ports of an *active* server establish the connection to the ports of a *passive* server and control the connection. Ports of a *passive* server *listen* for the connect requests of *active* ports and respond to the requests of the *active* port when the connection has been established. In a two party transfer, the client is one of the servers; it chooses to assume a *passive* role in this case (this minimizes the transactions: if it chose to be active, it would be necessary to request control or data port information from the passive server).

<u>IOD and IOR messages</u> Two primary data structures are exchanged between the client and servers: an *IOD* (IO Descriptor) which describes the function to be performed with the location of the data, and an *IOR* (IO Reply) message that describes the result of performing these functions (See **Data Structures**). For

the source server, the *IOD* contains the operation to perform (read), and the gather list specified by the user. For the receiving server (sink), the *IOD* contains the operation to perform (write), and where to put the data being received (the scatter list specified by the user). Each server may, in turn, contact other servers to perform the request. The actual transport function being performed may therefore involve a general network wide gather/scatter.

The client will begin, coordinate, and determine completion of the overall transfer. The control actions of the client consist of (See Fig. 7):

1. Sending an IOD to the passive server which describes the function it is to perform (read, write, other) and waiting for the IO reply messages that contain listen port addresses,
2. Sending an IOD to the active server to describe its functions, data locations, and the passive side port addresses to connect to derived from 1 (parallel transfer occurs at this point),
3. Waiting for IO reply messages from each server which describes the completion status of each side of the parallel transmission.

These control actions of the client are sequential; all other exchanges among the parties are parallel. For <u>two party</u> transfers, Step 1 above is performed internally by the client (the client simply readies a set of control or data ports for the parallel transfer in this case). Active server control and data ports will connect, control, and deliver the data in parallel once the setup is complete.

<u>Parallel Transport Protocol</u>

Initially all servers are listening on their public control ports. The client receives a user request and starts the transfer. The transfer may involve two or three parties. The client may be either of the servers or 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 performed internally within the client.

1. <u>***IOD*** sent to *Passive* Server.</u>

1.1 <u>The Client</u> generates a transaction identifier unique to this transfer which will be used in all messages exchanged among the parties involved in the transfer (this will be used to identify the group of parallel transactions as parts of a single logical transfer). One server is chosen to be passive; the other will assume the active role. For <u>Two Party</u> transfer, the Client will assume a passive role rather than an active role. ***Two party transfer is expected to be the predominant case in practice.***

<u>Three Party Transfer.</u> 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 transfer assignment IO distribution lists. If the passive server is the data source, then the gather list is sent, otherwise, the scatter list is sent.

<u>The client</u> waits for one or more replies (***IOR*** messages) from the servers assigned to the transfer (replies to the client's request may be coming from

other servers). When all of the data requested has been accounted for in the IORs returned to the client, the client can proceed (Part 2. below).

1.2 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 the device data to the ports 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.) 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 (step 3, 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 (See Example 1).

2. ***IOD*** Sent to Active Server.

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

- the relevant command to move data (read or write)
- a **SrcSinkList** containing 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,
- a **SrcSinkList** containing the logical transfer assignments of the user: a scatter (write) or gather (read) list as appropriate.

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

2.2 The Active Server receives the ***IOD*** describing the data move, reads the scatter(write) or gather(read) distribution list (which ever is relevant) and

- determines the physical media involved,
- determines direct device transfer capabilities of the media,
- generates control ports to maximize local move rates,
- maps the transfer window assignments to the control ports (assigns the control ports a portion of the transfer window)
- determines the port to port interconnect between the servers by reading the passive control port list transfer data window assignments supplied in the new **SrcSinkList** constructed by the client from the **IOR**s
- determines the data port protocol to be used on a connection basis

3. Connect Control and Data Transfer

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 3.4 below is performed (in this case it was found that no additional control exchanges would be necessary to setup 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.

3.1 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.

3.2 Determine connection control. Data ports are created or reused as necessary. An **InitiatorMsg** is sent to the passive server that contains

1) the data port protocol to use,
2) the transfer window data identification (**Woffset,Length**),
3) which side will assume control of the data connection, and
4) 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).

3.3 Passive control ports reply with an **InitiatorMsg** containing the data port address to connect to, and the **(Woffset,Length)** of the data to be moved. The amount of data moved between the data ports will be the minimum of the **Length**s exchanged between the two control ports (buffers may mismatch).

[If the active server delegated the active role to the passive side, then no message is sent -- the passive control port has assumed the active role. The active port supplied its data port address to connect to in this case; See Example 2, part 2].

3.4 Data Port Connect and Data Transfer:

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

3.4.2. 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.

3.4.3. Completion status is returned to the control port.

3.5 Iterate to 3.2 until all data transferred. Active control ports may optionally send a **CompletionMsg** to the respective passive control port giving the completion status and bytes moved before iterating.

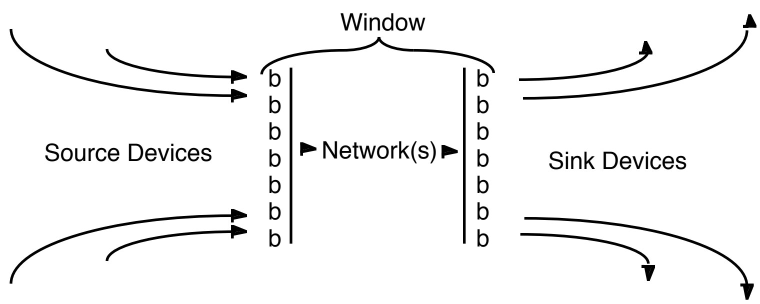
4. Termination. Each server collects completion status from the ports assigned to the transfer, forms a completion status list corresponding to the assigned

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.

The examples will develop the protocol detail as well as illustrate the use of the proposed data structures to be exchanged.

Conceptual Parallel Flow and Representation

Parallel flow will consist of a set of bytes all moving logically in parallel from devices of the sender to devices of the receiver. Bytes to be moved are 'gathered' from devices of the sender, form a logically contiguous sequence between source and sink machines, and are then 'scattered' to devices of the receiver.

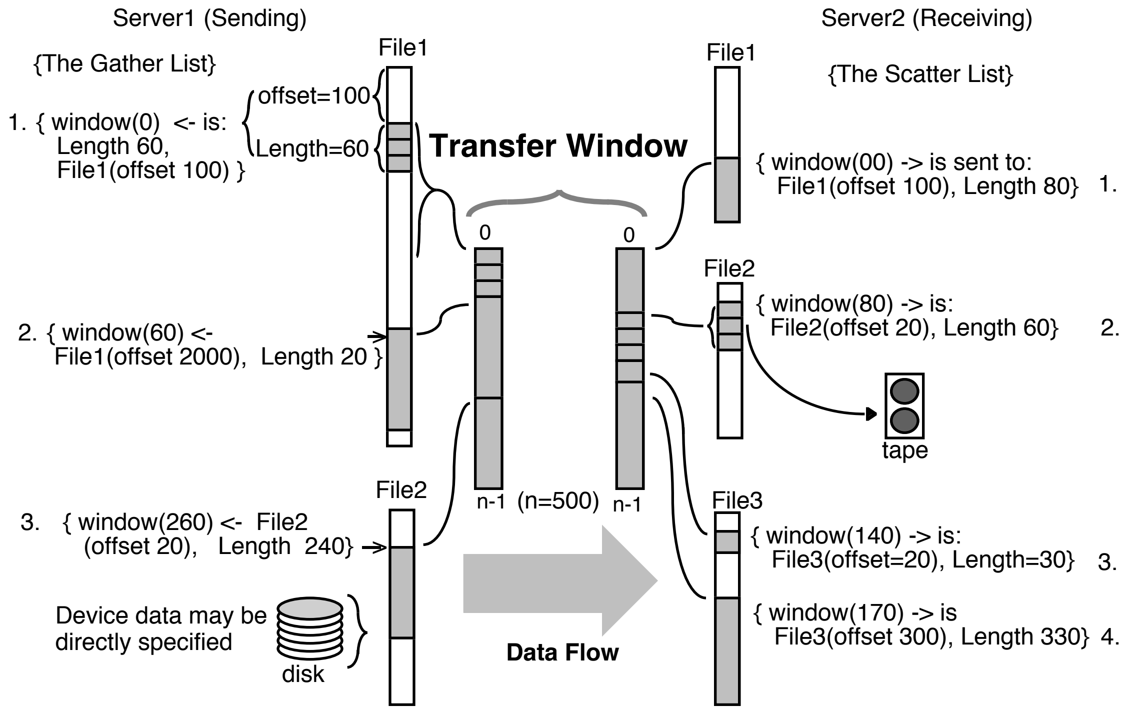


Parallel data flow consists of a set of bytes all moving conceptually in parallel from source devices to destination devices. These are 'gathered' from various locations by the sender and 'scattered' to various locations by the receiver. Any transfer can be considered to form an intermediate logically contiguous sequence of bytes (Window), that can be used to describe the data to be sent or received in a manner common to both sides of the transfer.

Fig. 8 Conceptual Parallel Flow

All bytes in a transfer will conceptually move in parallel from the sender to the receiver. The physical media involved in a transfer (e.g. disks, tapes, network interfaces, networks) may move portions of the data in sequential order without changing this view. A parallel transfer demands that a byte be uniquely represented in a move.

Objects that may serve as the source or destination of data may be any of the abstractions supplied by the local data management systems involved. These may be files in the traditional sense, local memory buffers, devices, ports or other addressable entities. The traditional view of a file as a logically contiguous sequence of bytes is maintained. Globally addressable direct access files are assumed where bytes may be accessed randomly.



In this example, data is being transferred between two file systems. A Gather list specifies the objects to be sent, and a Scatter list specifies the objects to be written. All bytes are unique, may move in parallel, and conceptually form a contiguous zero-based virtual address space transfer window between source and sink. Conceptually, data flows from local storage media to an offset within the window, through the window to the receiver, where the receiver moves the data from its window offsets to the local media. This transfer window is unique to each transfer, identified by a transaction identifier in all messages exchanged during a transfer, and is used to establish the correspondence among the many mappings involved between both sides of the transfer. Byte positions in the window are mapped to logical or physical addresses in the sender, the receiver, to data and control ports used for the transfer, and to the devices involved.

Fig. 9 Logical Representation and Correspondence Between Servers

An entire conceptual move as viewed by the user is represented as a single move request. The request may involve multiple blocks (a logically contiguous sequence of bytes) of data within multiple objects (i.e. files, memory buffers, devices, or network locations). A list of blocks describes a move; one for the sender and one for the receiver. Multiple objects do not add complexity once a general data selection methodology is defined.

The list of blocks in the sender, taken as a whole, describe where data is to be *gathered* from, and the list for the receiver describes where the data is to be *scattered* to. Block lists are similarly constructed for both sides of the transfer. The total number of bytes specified for each side must match and each byte must be uniquely represented.

List elements describe one or more logically contiguous sequence of bytes (blocks) of data. For each list element an object identifies a sequence of bytes appropriate for the context defined by the data management system involved. A file address, network address, device address, memory buffer location, a data base object, or other generic forms of address can be used to locate the object. For each address, there is a zero-based **offset** from the address to the first byte of the source or destination of the data to be transferred where there are **Length** bytes (see Data Structures: **SrcSinkList** definition).

The above figure illustrates the complexity of the transfers that can be formed. It may not typical for such complexity to be directly specified by an application. In parallel and distributed systems however, a simple logical request (a single block or stride over a file for example) that specifies a large amount of data, will almost always involve parallel storage devices. Increasingly, remote devices and other servers may be involved as well.

Design Alternative *A parallel transfer could have been viewed as forming a simple set of independent and unrelated transfers. The machinery required to represent the processing of each individual piece would lead to the same list structure and capabilities as are proposed above for an entire transfer i.e. the above structure would still have evolved for each individual piece. Each transfer must be logically related at some point. Treating a parallel transfer as a set of independent transfers moves this problem to a higher level. Optimization of the transfer is possible with the proposed machinery, and this would not be possible if the transfer were to have been split earlier. The client will not know the physical layout on the servers, and hence would not know how to partition a transfer appropriately.*

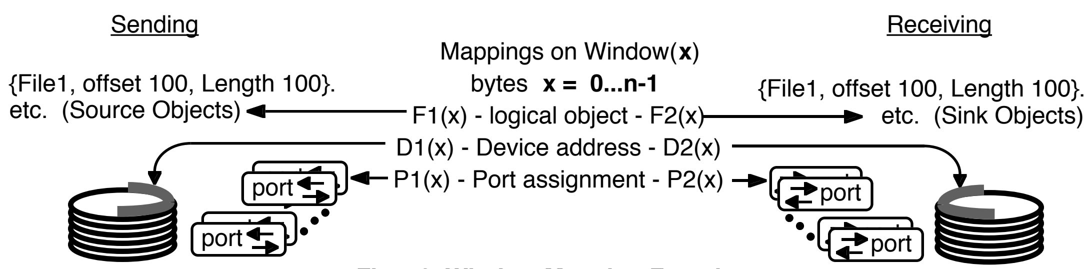


Fig. 10 Window Mapping Functions

Transfer *Window* Mappings

A transfer will require possibly many mappings on both sides. It is necessary for both servers to know their respective gather or scatter distributions as received from an application's request to move data, how these logical specifications map to local storage devices, and the mapping between the devices and ports. One server, the active server, will need to know the port mapping of the other. In order to maintain the correspondence between these and other mappings, the transfer is viewed as forming a single logically contiguous sequence of bytes between the servers, and data mappings are then based on the byte positions in this overall transfer ***window.***

Each list entry specifies a mapping of the bytes into a contiguous zero based virtual address space *window*. The address space has no physical counterpart, and is used only to describe a mapping of bytes to be moved from objects in the sender to objects in the receiver. For each list element, **woffset** is a zero-based byte offset within this *window* to a logically contiguous block of **Length** bytes formed by the list element. An ordering of the list elements is therefore not required.

Data Structures
Exchanged among Client and Servers

The protocol defines a set of data structures, their semantics for use between machines and a procedure for managing parallel data flow. New transport services and providers are not defined; existing services and providers need not be modified or extended. Realization of parallel transport is a matter of agreeing on a common representation of the control information to be passed between machines.

Two primary data structures are defined which are exchanged between client and servers: an IOD (IO Descriptor) which describes the service to perform on a set of data locations, and an IOR (IO Reply) which describes the result of processing this request. Additional data transport phase structures are defined which are exchanged between private control or data ports as necessary to control related data connections. An object addressing package is defined to locate objects in a manner relevant to a wide variety of environments.

The encoding of the proposed structures (i.e. packet forms for transmission) is not currently specified. Encoding for transmission will be analogous to DCE.

Portable basic types

Machine portable type definitions will be required. The following definitions are used in the proposed data structures.

```
/* Some Portable types */
typedef double floating;
typedef int int32;
typedef short int int16;
typedef unsigned short int uint16;
typedef unsigned long int uint32;
typedef unsigned long long int uint64;
```

Type int16 and int32 denote signed 16 and 32 bit integers while types uint16, uint32, and uint64 denote unsigned integers respectively etc. (a typical arrangement). A 64-bit unsigned integer designation is necessary.

All quantities will measure bytes unless noted otherwise.

The IO Descriptor

An IO Descriptor (**IOD**) is used to describe all IO requests. The **IOD** is sent to the public control port of each server involved in the transfer to begin and describe the transfer to that server.

```
                         /* The IO Descriptor */
typedef struct IOD{
    uint32          RequestID;          /* Transaction Identifier */
    uint32          Function;           /* The operation to perform */
    uint32          Flags;              /* Kind of IO request */
    address_t       Replyto;            /* Who to reply to (the client)*/
    uint32          SrcDescLength       /* Length of source list */
    uint32          SinkDescLength      /* Length of sink list */
    SrcSinkList_t   *SrcDescList;       /* The data to send */
    SrcSinkList_t   *SinkDescList;      /* The data to receive */
} IOD_t;

/* Values for Function and Flags fields in IODs */
enum IOfunctions {IOread=1,IOwrite=2};
```

RequestID is the transaction identifier, unique to the series of requests and replies across the servers involved, is used in all messages related to the transfer, and serves to identify the series of transactions as part of a single request.

Function is the operation to perform. For the sending server the **Function** is **READ**, and **SrcDescList** describes the data to be gathered. For the receiving server, the **Function** is **WRITE** and **SinkDescList** describes the data to be scattered.

Flags is a bit vector describing the type of **IOD**, how to process the **IOD** and what type of reply is desired. These values may also be carried to the **Flags** field of the **SrcSinkList** (see below) to describe each portion of the transfer.

Replyto the control port address that the **IOR** should be sent to (the server sending the reply may not be the server who was sent the **IOD**; the server sending the reply needs to know who to send the reply to). **Replyto** is set for the intermediate reply (**ReplyWithPorts**=1), otherwise (**ReplyWithPorts=0**,**Replyto**=null) the final IOR is a completion status message which is always sent to the party that sent the **IOD: Replyto** is irrelevant.

SrcDescLength and **SinkDescLength** are the lengths of the source and sink descriptor lists respectively. **SrcDescList** is a list of blocks (the gather list) that together describe the source of the data, and their respective position in the overall transfer window. Similarly, **SinkDescList** is a list of blocks (the sink list) that describes where the data is to be sent, and where it occurs in the overall transfer window (see source sink list definition below).

```
    /* Values for the Flags field in IODs and SrcSink Descriptors.*/
#define signed32 SequentialIO       = 0x00000001;
#define signed32 ReplyWithPorts     = 0x00000010;
#define signed32 ControlAddr        = 0x00000020;
#define signed32 Responder          = 0x00000040;
#define signed32 LastInXfer         = 0x00000080;
#define signed32 XferIP             = 0x00000100;
#define signed32 XferIPI3           = 0x00000200;
#define signed32 XferProtocol1      = 0x00000400;
#define signed32 Xfer etc.          = 0x00000800;
```

SequentialIO The data being described must be sent in the order requested. It applies to the entire request. If set, a sequential device such as a tape or frame buffer is involved in the transfer. Otherwise the data may be sent in any order which optimizes resource utilization.

Sequential devices involved in a transfer impose a preferred or required ordering on the data to be transferred. Frame buffers must either request or be sent data in order. Tapes are not seekable on writes. For writing tapes, out of order delivery would require a buffer the size of the entire transfer to provide for the first record arriving last (parallel transport expects the movement of large data sets). A tape system is given the ability to control its portion of a transfer and *pull* data from other systems in the order required to write the tape if this option exists.

ReplyWithPorts This flag indicates that Server1 is to be passive for this request. An intermediate reply to the **IOD** should be returned which contains a control (or data) port list with their transfer window assignments; listens should be posted on those ports. The addresses in the **PortList** returned by the **IOR** will be the port addressing information for the active server to connect to.

ControlAddr: is set in an intermediate IOR and IOD if the address is a (network) control address rather than a data port address.

Responder: An *'Initiator'* makes a request, and a *Responder* drives a data connection. **Responder** is set to indicate 3rd party capable (e.g. IPI-3 conforming) devices are involved in the portion of the transfer described by the source or sink descriptor list element. Each server will determine which devices involved in portions of the transfer are 3rd party capable (i.e. capable direct transfer), and if so, will set this flag.

The device may control a part of the transfer in this case, rather than the server, if the protocol in use supports device to device transfers (Intelligent Peripheral Interface-IPI, or equivalent protocol). The active server will choose the *Responder* based on this flag. If an active *Responder* is present (on both sides, or on the active side), then the active control port controls as usual. If the *Responder* is only on the passive side, then the *Responder* will assume an active control role over the connection (i.e. active/passive data port roles will be reversed: the data port on the passive side will issue the connects and push or pull the data, while the data port on the active side will post the listens. See Example 2 case 2)

LastInXfer is set if the last byte in the transfer is present. This is device dependent information, (which some devices require).

XferIP, **XferIPI3** etc. are the high level protocol options of the server. These flags are set by the passive server to convey its data transfer options to the active server. If **XferIPI3,** a data port is IPI-3 capable (direct device, or direct device to device transfers are possible if the devices support IPI-3). The active side of the transfer will chose the appropriate data protocol to use based on what is available to both sides on a port to port basis. **XferIP** is set if IP can be used etc.

```
                    /* Source/Sink List definition */
typedef struct SrcSinkList {
    uint64        Flags;          /* How to treat this data */
    uint64        Woffset;        /* Location of data in the transfer */
    uint64        Length;         /* Number of logically contiguous bytes  */
    address_t     AnAddress;      /* Where to get or send the data */
    struct SrcSinkList *Next;/* next list element */
} SrcSinkList_t;

SrcSinkList_t GatherList, ScatterList;
```

<u>**SrcSinkList**</u> is a list of 1 or more entries which describe where to locate the data, and how it relates to the overall transfer (i.e. where it fits in the window). **Woffset** is the relative position of the data described by this element in the overall transfer ***window,*** and **Length** is the amount of data in bytes at this offset. Each list element may be positioned in the window independent of its order in **SrcSinkList** by appropriate values of **Woffset.**

AnAddress describes where to locate the data to be sent or received, and may be any one of many different types of addresses (e.g. a file address, a network address, a device position, etc.). An address offset is typically associated to locate the data (see **address** definition).

The IO Reply

One or more reply messages, **IOR** messages, are returned as the reply to each **IOD**. The **IOR**s returned contain intermediate setup information (control or data port addresses and their data assignments), and final completion status and error information to locate any transmission or device errors.

```
                    /* The IO Reply */
typedef struct IOR{
    uint32          RequestID;          /* Transaction Identifier */
    uint32          Flags;              /* Kind of IOR */
    uint32          Status;             /* If error, kind of error */
    enum Reply_t    ReplyType;          /* Kind of reply */
    PortReply_t     *PortList;          /* List of control port addresses */
    uint32          SrcReplyLength;     /* Src reply length */
    uint32          SinkReplyLength;    /* Sink Reply Length */
    SrcSinkReply_t *SrcReplyList;       /* Reply List for source descriptor list*/
    SrcSinkReply_t *SinkReplyList;      /* Reply List for sink descriptor list */
} IOR_t;
enum Reply_t {IOlistenList, IOsrcsink,...}; /* IOR Reply Types */

              /* Source/Sink Reply Definition */
typedef struct SrcSinkReply {
     uint32          Flags;              /* If error: */
     uint32          Status              /* Kind of error */
     uint64          BytesMoved;         /* Bytes Moved Successfully*/
     [ptr] struct SrcSinkReply *next
} SrcSinkReply_t

         /* Flag values for the IOR and SrcSinkReply */
#define uint32    IOcomplete    =2;     /* Complete w/o error */
#define uint32    IOerror       =4;     /* error detected */
#define uint32    etc.
                    /* Port List Definition */
typedef struct PortList{
    uint32          ListLength          /* Length of port list */
    SrcSinkList_t *ListenList;          /* List of control ports and assignments */
} PortReply_t;
```

RequestID is the unique transaction identifier used in all messages during the transfer to identify and verify the messages as part of an overall transfer.

Flags determines how to interpret the **IOR**. **IOcomplete** is set if the server is finished processing the request. **IOerror** is set if an error was detected during processing the IOR. The particular error or errors will be given by the **Status** field of the **SrcSinkReply** list.

Status is the result of processing the request as a whole. A value other than zero indicates an error has been encountered when processing the **IOD**.

ReplyType is the type of the reply. **IOlistenList** indicates a port list is being returned (rather than **Src/SinkReplyLists**).

SrcReplyList is a list of responses for each element in the **SrcDescList**, and **SinkReplyList** is a list of responses for each element in the **SinkDescList.**

These are mirror images of the IOD. **SrcReplyLength** and **SinkReplyLength** are the length of these lists respectively.

SrcSinkReply describes the result of processing the respective entry in the **SrcSinkList**. **BytesMoved** is the number of bytes successfully moved prior to an error if any.

Flags is a bit vector identifying the completion type of a particular entry in the **SrcSinkList**. **IOcomplete** is set if processing of the entry completed, and **IOerror** is set if an error was detected during processing. The particular error encountered is defined by the **Status** field.

Status. A value other than zero indicates the type of error.

PortList is a **SrcSinkList** of control or data port addresses with their data assignments. For each entry in this list, the address part is the control port network address, and **Woffset** and **Length** describe the position in the overall transfer that this port will be responsible for sending or receiving. **Flags** contain the protocol options at this port. This list is returned in response to **ReplyWithPorts** of the **IOD**.

Data Transport Phase Structures

Initiator and Completion Messages. Control ports may be associated with one or more data ports when additional control information is necessary to set up the data transfer. One or more **InitiatorMsg**s are exchanged between each pair of control ports to set up and control the data exchange. This is done just prior to data transfer.

```
                        /* The InitiatorMsg */
typedef struct InitiatorMsg{
    uint64        Delimiter;     /* A unique identifier */
    uint32        Flags;
    uint64        Woffset;       /* Location of data in the transfer */
    uint64        Length;        /* Number of bytes to transfer */
    uint64        TotalLength;   /* SrcSinkList entry length */
    uint32        AddrType;      /* kind of network address to ConnectTo */
    address_t     ConnectTo;     /* The address to connect to (if not Responder) */
    char    SecurityTicket[8];   /* Security info */
    uint64        CheckSum;      /* of the message */
} InitiatorMsg_t;
                        /* The CompletionMsg */
Typedef struct CompletionMsg {
    uint64        Delimiter;     /* A unique identifier */
    uint32        Flags;
    uint32        Status;
    uint64        BytesMoved;    /* the number of bytes moved */
    char    SecurityTicket[8];   /* Security info */
    uint64        CheckSum;      /* of the message */
} CompletionMsg_t
                        /* Flag values */
#define Responder          1     /* sender will be assume Responder role */
#define NeedCompletion     2     /* Initiator needs completion message */
```

InitiatorMsg is exchanged between (private) control ports to supply set up and control information for the actual data exchange between data ports. A **RequestID** is not [currently] needed because the control ports involved are created unique to each portion of the transfer.

Delimiter is used to verify that this is an **InitiatorMsg.** The peer control port will know to expect the **InitiatorMsg** and will check the **Delimiter** to ensure that one being received when expected. **Woffset** is the location in the transfer identifying the data to be sent or received. **AddrType** is the type of network address to connect to (**ConnectTo**). A **CheckSum** of the header is performed to guarantee integrity.

Length is the number of bytes to be transferred. **TotalLength** specifies the amount of data contiguous within the transfer window that the active server will control using the same data transfer mechanism (this allows for additional optimization by the passive server in the case where 3rd party transfers are possible).

Flags values are: **Responder=1** if active side data ports are to control the connections; passive side data ports will issue the connects otherwise (An *Initiator* makes a request and the *Responder* controls the data port connection). **NeedCompletion=1** if the initiator requires a completion message.

CompletionMsg is optionally returned by the *Responder* when data transfer is complete. **Status** is completion status information. **BytesMoved** are the total number of bytes moved successfully over the connection. The remaining fields are the same as in **InitiatorMsg.**

Data Port Data Tag. When sending and receiving data over a data connection, a method of identifying each piece of data sent over the data connection is needed. The data tag defined here will be used if an equivalent is not available in the protocol used for the transfer (e.g. if **IP** is being used, then the data tag defined here would be needed but if the **IPI-3** device to device data transfer protocol is being used, then the data tag defined here would not be necessary since the information is already embedded in the **IPI-3** protocol header).

```
                    /* The Data Tag */
typedef struct {
    uint64      Delimiter;    /* Unique Identifier */
    uint64      RequestID;    /* Transaction Identifier */
    uint64      Woffset;      /* Location of data in the transfer */
    uint64      Length;       /* Number of bytes in block of data */
    uint64      CheckSum;     /* of the data tag */
} DataTag_t;
DataTag_t DataTag;
```

DataTag is used to identify each block of data sent or received over a data connection. A 64-bit constant **Delimiter** verifies a **DataTag.** The data port will know to expect the **DataTag** and will check the **Delimiter** to ensure that a DataTag is being received when expected. **RequestID** is the unique transaction identifier identifying the address space (window) for the transfer. **Woffset** and **Length** give the window position of the data to describe where the data is relative to the transfer as a whole. A **CheckSum** of the header is performed to guarantee integrity.

Data Tag Usage. The **DataTag** will be used if necessary to either ***Push*** data to, or ***Pull*** data from the peer port over the connection as follows:

Push send the **DataTag** to the passive data port to describe the data being sent e.g. {Woffset, Length}, followed by the data.

Pull send the **DataTag** to the passive data port to request the data {Woffset, Length}. Passive port replies with **DataTag** followed by the data.

Addresses

These structures define addresses that the protocol will need to communicate the location of data between servers. These addresses are used in the source/sink descriptors to define the sources and sinks of the data transfers. The address and offset in the source/sink descriptor may change as the request is processed to reflect the location of objects, ports, or device locations as necessary. A generic address will be defined in a general framework that can be expanded over time as needed. The number and kinds of addresses can be expected to increase.

Data transport between machines should provide a method of referencing the objects involved in a manner acceptable to all platforms, independent of which data management system is involved. An application requesting a parallel data transfer, will know how to refer to the abstractions managed by the various data management systems being used. The impact on the protocol is in developing an object addressing package that is sufficiently general to represent the 'address' of any of the objects managed by the various data management systems involved in the manner suitable to those systems.

Object Addresses

```
typedef struct {
    char        ObjectName[];     /* text name of data */
    uint64      ObjectOffset;     /* offset to data */
} objectaddress_t;

typedef struct {
    handle_t    Handle;           /* returned by open or lookup */
    uint64      HandleOffset;     /* offset to data */
} handleaddress_t;
```

Object addresses serve to locate data in a manner specific to a particular data management system. The user of a particular data management system will know how to refer to the objects managed by those systems. The protocol passes the addresses supplied by the user on to the servers involved.

Object addresses are readable (e.g. a character string as in a UNIX path name or a data base object name), or non readable (e.g. a handle returned as the result of opening a file, or the result of a look up in a data base).

objectaddress_t defines a human readable form. **ObjectName** is the text name of the object in the environment (e.g. a path name in a Unix file system). **ObjectOffset** is the zero-based origin to the data.

handleaddress_t defines a non readable form of address (e.g. returned as the result of opening a file in a file system environment), and **HandleOffset** the zero-based origin to the data.

Network addresses

```
typedef struct {
    uint32          RequestID;
    uint64          SockOffset;
    struct {int32 addr,port,family;} SockAddr;
} IPaddress_t;

typedef struct {
    uint32          RequestID;
    uint64          IPI3Offset;
    struct { int16 Interface;
        Char Name[IPInamesize]; } IPI3Addr;
} IPIaddress_t;

ETC.
```

IPaddress_t defines an IP address. **RequestID** is the transaction identifier used to identify and verify this portion of the transfer. **SockOffset** is the zero-based byte offset at the port (this is the offset that will be used in the DataTag and is expected to be the window offset) **SockAddr** contains the IP address information for the socket: **addr** defines machine, **family** defines type of transfer (e.g. TCP), **port** defines which process to connect to on a particular machine.

IPIaddress_t defines an IPI-3 address. **RequestID** is the transaction identifier used to identify and verify this portion of the transfer. **IPI3Offset** is the zero-based byte offset at the port (this is the offset that will be used in the DataTag and is expected to typically be the window offset) **IPI3Addr** is the IPI3 specific addressing information: **Interface** defines the low level protocol (e.g. HIPPI, Ethernet), and **Name** is used to lookup the routing information (a HIPPI I-field for example).

Device addresses

Direct device references may be introduced by mass storage system interface requirements. Implementations which need to reference physical devices will need to negotiate precise definitions for generic device addresses.

```
                    /*  Device addresses */
typedef struct devaddress{
    int32 DeviceId;
    char VolumeId[somesize];
    positiondesc_t DevicePosition;
} devaddress_t;

typedef struct{
    char *ptr [];     /* pointer to data in memory */
} memaddress_t;
```

devaddress_t an example device address. **DeviceId** is the device identification as designated by the server. **VolumeID** is the media identification for verification of the device, and **DevicePosition** is the origin of the read or write on the media.

memaddress_t a memory buffer: a pointer to a sequence of bytes. A buffer is usually tied to a network address.

A Generic Address

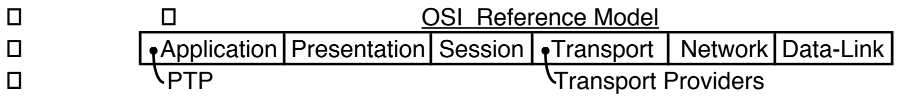
```
enum AddressTypes{IPI,IP,OBJ,HAN,DEV,MEM,etc} ;

typedef struct {
    enum AddressTypes AddrType;
    union {
        IPIaddr_t           IPIaddr;      /* An IPI address */
        IPaddr_t            IPaddr;       /* A Socket address */
        objectaddress_t     ObjectAddr;   /* text e.g. a Unix path name */
        handleaddress_t     HandleAddr;   /* binary e.g. an opened file */
        devaddress_t        DevAddr;      /* a device specific address */
        memaddress_t        MemAddr;      /* a memory buffer */
    } Addr_u
                                        } address_t;
```

address_t defines the location of the source or destination of the data. A generic address constructed as the discriminated union of all the possible types of objects that may serve as the source or sink of data.

Transport Level Services Required

The Parallel Transport Protocol (PTP) will be a Transport Layer Application which will be implemented with standard Transport Interfaces to existing Transport Providers (protocols which supply reliable cost-effective data transport services).



Any transport provider which supplies reliable delivery may be used for implementation. TCP is an example of a reliable transport provider. **Any** Transport Interface supplying a connection mode service may be used (e.g. Berkeley ***Sockets***, UNIX Transport Layer Interface (***TLI***), others). It is not relevant which interface is used as long as the services required are supplied.

The transport layer is the lowest layer on which reliable end-to-end data delivery is defined; a parallel transport definition cannot be defined lower than this without modifying the definition of networks and their interfaces.

Transport Services A connection oriented service is required which provides a sequenced, byte-stream data delivery over flow controlled, optionally buffered connections that guarantee reliable delivery using time-out and retransmission or equivalent. These are services that are commonly supplied; *they are not specific to parallel transport.*

Connection Oriented PTP requires a connection oriented service. Connection-oriented services require that two conversing applications establish a logical connection between each other for communication. This involves: 1) connection establishment, 2) (lengthy) data transfer, and 3) connection close. **Sequenced** delivery is required when used on packet-switched networks (e.g. Internet. On packet switched networks, data transmitted on a connection may be divided into *packets* which may arrive out of order; the protocols used must provide sequencing to guarantee that data transmitted over a connection is received by the receiver in the same order that it is transmitted by the sender).

Reliable Delivery PTP does not define error recovery on a connection basis: if an error is reported on one or more connections, PTP reports failure back to the client. Error free data delivery is expected to be a property of the protocols in use. We expect use of **checksums** to detect data corruption, **retry** to resend any corrupted packets detected, use of **positive acknowledgment** to acknowledge receipt, **time-out** with resend to detect and recover from loss of positive acknowledgment, and **duplicate detection** to recover from multiple copies of the same packet being sent during error detection and recovery.

Buffering may be used on a connection (there are no interactive requirements in the parallel transport machinery). End-to-end **Flow Control** is required, to pace the flow of data between the sender and receiver processes exchanging data, (so that the sender does not send data faster than the receiver can accept it). A **Byte-stream** service must be used (no record boundaries in the data stream).

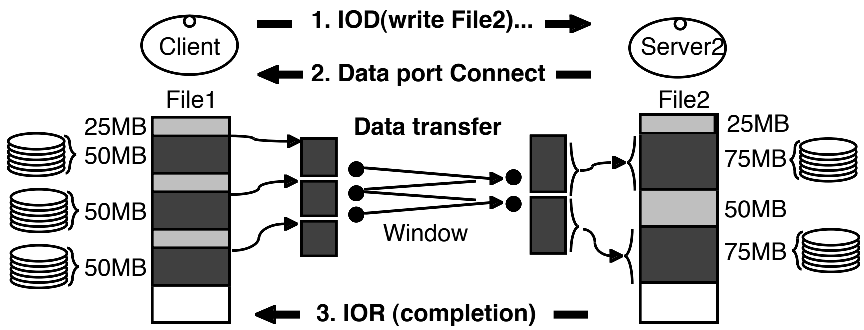
Example 1

Simple Two party transfer

This example details the interactions necessary to transport data in parallel between a client and a single server where no additional private control exchanges are necessary to set up the transfer on each data connection. This is a highly specialized but frequently occurring case.

A Client decides to move 150MB from File1 (which it owns), to Server2 File2. Assume the data is in 50MB blocks on devices which the Client controls, and is to be written to File2 in 75MB blocks. For both servers, suppose each block is on separate devices which may run in parallel (i.e. have separate or high speed paths to external networks).

Client will be passive sender **Server2 will be Active receiver**



A Client decides to write data (which it owns) to Server2. The data is on three devices which can deliver the data in parallel. Suppose the client determines that three data ports can be used to maximize local transfer rates. The Client builds an IOD containing its data port addresses, the data to be written on File2, and sends this write request to Server2. When the request is received, Server2 allocates two ports to maximize its local transfer rates. Server2's data ports connect, **in parallel**, to the data port addresses supplied in the Client's IOD, and transfer the data **in parallel**. A completion reply message is returned to the Client.

The assumptions being made here are that no additional control exchanges are necessary to move the assigned data: the data is known to fit available buffers, and the protocol to be used is known by the client. Data, rather than control, port addresses can be passed to Server2. (These are the same assumptions that would be made by other network transport facilities such as NFS.) This is a simple but common case.

Fig. 11 Two Party Transfer (without connection control)

A transfer is initiated by the user. The Client API determines how to refer to the objects involved (e.g. determines the object addressing form, opens files, derives or supplies the handles, etc.). The Client API specification is not part of this protocol.

A gather list is composed by the user application to specify where the data is coming from and moving to. In this case, the Gather and Scatter lists are:

GatherList :

[File1, offset 025MB, Length 50MB] -> (Woffset 000MB)
[File1, offset 100MB, Length 50MB] -> (Woffset 050MB)
[File1, offset 175MB, Length 50MB] -> (Woffset 100MB)

ScatterList:

[File2, offset 025MB, Length 75MB] <- (Woffset 00MB)
[File2, offset 150MB, Length 75MB] <- (Woffset 75MB)

This transfer is a random list of blocks which, taken together, form a logically contiguous 150MB transfer. Each block is assigned a position in the overall transfer (Woffset). Above, for example Woffset is assigned in the order of the blocks in the list. If parallel transfer is appropriate, the Client API calls the transfer routine e.g.

Transfer(Client,GatherList, Server2,ScatterList)
(Not part of the protocol)

Window Definition The gather list defines a mapping of the logical objects of sender onto a logically contiguous 150MB transfer window. Similarly the scatter list defines a mapping between the window and the recipient objects. The window is not actually formed: it is a method used to establish the correspondence of transfer data mappings for both sides of the transfer.

The correspondence of the data being sent to logical data being received for this transfer is as follows:

	(Sending)				(Receiving)	
Object	(offset)	Length	Woffset	Object	(offset)	Length
File1	025MB	50MB	000MB	File2	025MB	050MB
File1	100MB	25MB	050MB	File2	075MB	025MB
File1	125MB	25MB	075MB	File2	150MB	025MB
File1	175MB	50MB	100MB	File2	175MB	050MB

A given byte in the transfer is referred to by its position in the overall transfer window. Woffset gives the position of a piece of data in the overall transfer window. We can refer to the window position of a byte in the transfer to establish the correspondence between its logical addresses, its location on devices, and at ports.

1. Passive IOD

The Client assigns a unique transaction identifier to identify all dialogue related to subportions of this transfer: RequestID=**F1F2Move.** The Client always chooses itself to be passive, whether or not it is sending or receiving (otherwise it would need to request control port addresses from Server2). A passive side of the transfer only needs to know how to move its local data since the active side will connect to the port addresses supplied by the passive side and control the delivery or receipt of each portion of the transfer.

Client:

1. Performs the logical to physical device mapping (i.e. determines what devices are involved in moving the data). Since the Client's **Function** is **READ,** the **GatherList** is used to determine the logical objects involved, and the data management system supplies the physical device allocation.
2. Determines the ports that will be used to transfer the data. Assume data ports are allocated as necessary to maximize local transfer rates.

   We are assuming that no additional control dialogue is necessary to transfer the assigned data; data ports can be immediately assigned to the transfer.

   The data port protocol to be used must be determined by the client in this case.
3. Maps the data as stored on physical media to the data port which will be used for its transfer.
4. Posts listens on each of the data ports.

2. Active IOD

The Client constructs an IOD for Server2 containing its chosen port assignments, the logical data layout for the active side, sends the IOD to Server2, and waits for a termination status reply (**IOR**) from Server2:

Client IOD for Server2

```
IOD:{RequestID:    F1F2Move,
        Function:     IOwrite,
        Flags: ,
        SrcDescLength:3, SinkDescLength:2,
        ReplyTo: reply to sender with final IOR,
        SrcDescList:
            {Flags:SeeBelow,Woffset:00MB, Length:50MB,
                AnAddress:{AddrType:IP, Port1a}}
            {Flags:SeeBelow,Woffset:50MB, Length:50MB,
                AnAddress:{AddrType:IP, Port1b}}
            {Flags:SeeBelow,Woffset:100MB, Length:50MB,
                AnAddress:{AddrType:IP, Port1c}}
        SinkDescList:
            {Flags: ,Woffset:00MB, Length:75MB, Next: ,
             AnAddress:{AddrType:HAN, Handle:File2,HandleOffset:025MB}}
            {Flags: ,Woffset:75MB, Length:75MB, Next:null,
             AnAddress:{AddrType:HAN, Handle:File2, HandleOffset:150MB}}        }
```

The IOD for Server2, the active server, contains:

1. The Request Identifier **F1F2Move**
2. The function to perform: **WRITE**
3. **Flags** are

   ControlAddr=0 since a data port address is present,
   Responder=0 since the data ports will assume a passive role.
   ReplyWithPorts=0 since no reply is necessary.
   the data port protocol to use (**XferIP, XferIPI3,...**) determined by the client.

4. The length of the source and sink descriptor lists.
5. Who to reply to (**Replyto**) is null, since the final **IOR** will be sent to whomever the **IOD** was received from.
6. The source descriptor list containing the data port lists of the Client.
7. The **SinkDescList** specifies where Server2 is to write the data from the transfer window e.g. the first entry in this list specifies that **Length=75MB** is to be written to **File2 Offset**=**25MB** and that the data will be coming from **WOffset=0** of the transfer.

Server2 receives the request from the Client. Server2 determines device data layout from the **SinkDescList**, assigns data ports responsible for moving the data, and maps physical device data to these ports (i.e. the same processing that the client performed when reading the Gather list described above). Suppose the data is stripped on two devices **75MB** on Device2a and **75MB** on Device2b (see figure above). Suppose that these devices may potentially operate in parallel and that two ports {Data port**2a,** Data port**2b**} can to be used to move the data in parallel and maximize local transfer rates.

Determine data port interconnect. Server2 can now determine the port to port interconnect and begin the transfer. When Server2 has determined its local data ports and their data assignments, the passive port information (**SrcDescList** in this case) is read to determine the interconnect. Since active Data port**2a** is responsible for **75MB**, and Data port**1a** is responsible for **50MB** at window offset 0, Data port**2a** will connect to Data port**1a** for **50MB**, and then to data port**1b** for **25MB** etc. The port to port interconnects will in general be the cross product of the ports in the transfer. In this case, the interconnects will be:

Data port**1a** <-> Data port**2a**	woffset 0,	length 50MB
Data port**1b** <-> Data port**2a**	woffset 050MB,	length 25MB
Data port**1b** <-> Data port**2b**	woffset 075MB,	length 25MB
Data port**1c** <-> Data port**2b**	woffset 100MB,	length 50MB

3. Connect, and Data Flow

In the simple special case being assumed for this example, there is no connection control determination necessary between control ports, and data port addresses were passed by the Client. Data ports will simply connect and transmit data. Example 2, will detail connection control.

The data port ***Push/Pull*** algorithm will be performed if necessary on a port to port basis using the **DataTag** and procedure defined in the protocol data structures:

Push send the **DataTag** to the passive data port to describe the data being sent e.g. {Woffset **0**, Length **50MB**}, followed by the data.

Pull send the **DataTag** to the passive data port to request the data e.g. {Woffset **0**, Length **50MB**}. Passive port replies with **DataTag** followed by the data.

Otherwise, the mechanism supplied by the transport protocol chosen would be used to ***Push*** or ***Pull*** the data across the connections.

Server2 is Active and will therefore control the transfer:

Port2a.	Connect to Port1a,	***Pull***	Woffset 00**MB**,	Length **50MB**
Port2a.	Connect to Port1b,	***Pull***	Woffset **50MB**,	Length **25MB**

and In parallel,

Port2b.	Connect to Port1b,	***Pull***	Woffset 0**75MB**,	Length **25MB**
Port2b.	Connect to Port1c,	***Pull***	Woffset **100MB**,	Length **50MB**

*Suppose Server2 had been chosen to be passive. Then Server1 ports would be controlling the transfer and **Push**ing the data across:*

Port1a. Connect to Port2a, ***Push*** *Woffset 0,* *Length* ***50MB***
etc.

4. Completion IOR

Server2 Replies with Completion status to Client:

IOR**{**RequestID,
Flags: **IOcomplete**,
Status: **0** *(no error; for error definitions, see **Errorno.h**)*
PortList: **Null,**
SrcReplyLength **3**, SinkReplyLength **2**
SrcReplyList
{Flags:**IOcomplete**, Status:**0**, BytesMoved:**50MB**}
{Flags:**IOcomplete**, Status:**0**, BytesMoved:**50MB**}
{Flags:**IOcomplete**, Status:**0**, BytesMoved:**50MB**}
SinkReplyList *(what we thought we delivered to the other side)*
{Flags:**IOcomplete**, Status:**0**, BytesMoved:**75MB**}
{Flags:**IOcomplete**, Status:**0**, BytesMoved:**75MB**}
Send(Client, IOR); receive(); *client is listening for IO status returns*

The **Src/SinkReplyList**s mirror the information in the **IOD Src/SinkDescList**s. The client will know how to locate errors in the overall transfer by relating it to the IOD that was sent.

The interpretation of **SrcReplyList** is how much information Server2 received from the passive server over the connections, and the interpretation of **SinkReplyList** is how much information Server2 successfully wrote.

Termination. Ports are closed. The client returns completion status to the API when the IOR is received from Server2.

Example 2

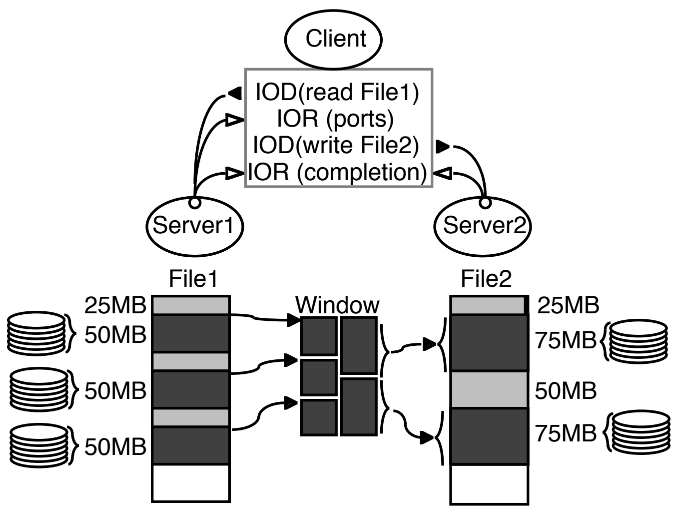
Full Protocol (Connection Control)

Two or Three Party Transfers

This example details the interactions involved in a two or three party transfer which involves connection control (all cases other than Example 1). Additional control exchanges will be necessary on private control connections to setup the transfer over the related data connections. For the two party case involving connection control, the actions described below for the passive server are assumed by the client and no external messages are sent.

A client decides to move 150MB from Server1 File1 to Server2 File2. The client is a third party in this transaction. The data to be moved is in three 50MB blocks on Server1, and is to be written to File2 in two 75MB blocks. For both servers, assume each block has been allocated to separate devices which may run in parallel (i.e. have separate or high speed paths to external networks).

Server1 will be Passive sender **Server2 will be Active receiver**



A Client decides to move 150MB from Server1 File1 to Server2 File2. The Client is a third party. The data to be moved from Server1 is stripped on three devices 50MB apiece, and the data to be moved to Server2 is stripped on two devices in 75MB blocks. Each server strives to maximize overall local transfer bandwidth. Suppose all devices may move data independently. Server1 allocates 3 control ports to control the movement of the data locally, and Server2 allocates 2. Server1 will be passive (Control ports listening for connects), while Server2 will be active (control ports issuing the connects and driving the connections).

Fig. 12 Full Protocol; Two or Three Party Transfers

1. Passive IOD

The Client assigns the unique transaction identifier to identify all dialogue related to subportions of the transfer: RequestID=**F1F2Move.** The Client uses the gather list supplied to build the **SrcDescList** of the IO Descriptor to describe the passive side of the move. The Client chooses Server1 to be passive in this transaction, and will request that Server1 return a new **SrcDescList** containing the control ports, their transfer window assignments, and protocol options, so that this can be sent to Server2. The passive side of the transfer only needs to know how to move its local data so **SinkDescList** will be null in this case (since Server1 will **READ** data rather than **WRITE** data). The Active side will connect to the control ports returned by the passive side to control the delivery or receipt of each portion of the transfer. The client connects to Server1 public control port (which is listening on its well known control port for client connects), and sends the IOD:

<u>IOD for Server1</u>

```
IOD:{
        RequestID:    F1F2Move,
        Function:     IOread,
        Flags:        ReplyWithPorts:1,
        ReplyTo:      {AddrType: IP, IPaddr{addr: Client IP address}},
        SrcDescLength: 3, SrcDescList:
        1 {Flags: ,WOffset:0MB, Length:50MB, Next:2,
            AnAddress:{AddrType:HAN, Handle:File1, HandleOffset:025MB},
        2 {Flags: ,WOffset:50MB, Length:50MB, Next:3,
            AnAddress:{AddrType:HAN, Handle:File1, HandleOffset:100MB},
        3 {Flags: ,WOffset:100MB, Length:50MB, Next:null,
            AnAddress:{AddrType:HAN, Handle:File1, HandleOffset:150MB}  }
        SinkDescLength:0, SinkDescList:null}
```

The IOD for Server1 has:

1. The Request Identifier, **F1F2Move** which identifies all transactions for this IO request.
2. The function to perform, **READ** in this case
3. Flags **ReplyWithPorts=1** to indicate that Server1 should ready a set of control ports and reply with these ports in an intermediate **IOR**. This flag also indicates that Server1 is to be passive for this request.
4. **ReplyTo.** The (Client) network address that the **IOR** messages are to be sent to (the party replying with the **IOR** need not be the same one that was sent the **IOD** and will need to know whom to send the reply to).
5. **SrcDescLength=3 -** The length of the source list that describes the logical data layout and its position in the window.
6. The **SrcDescList** for reading the data. This is the gather list that specifies how to locate the data, and where it is in the overall transfer (i.e. where it is in the transfer window). The transfer window implied is a logically contiguous **150MB** block at **Woffset=0**.
7. **SinkDescLength:0** The sink list is null because Server1 is a passive source server and does not need information about the active server: there is no need to include the sink data description. The **IOD** is always sent to the passive side first: one of the lists will always be null. If the Sink side of the transfer had been passive, then **SrcDescLength** would have been 0.

<u>CLIENT waits for replies.</u> The Client may receive possibly many replies to this request as control ports assigned to the transfer reply from various network locations. The Client knows when all replies are received when total length returned is what was requested. The Client is looping, waiting for replies:

```
Send(IOD,Server1ControlPort)
DoWhile length > 0 {
        receive();
        If an IOR with this RequestID is received, then
                process the IOR:
                (Build Server2's IOD)
                length=length-sum(Length) }
```

The IOD built for Server2 will contain the control port lists returned (the client requested this with **ReplyWithPorts**). The Client will send these to Server2, so that it can connect and control the data transfer.

1.1. Server1 Replies with ports

Server1:

1. Performs the logical to physical device mapping (i.e. determines what devices are involved in moving the data). Since Function is **READ,** the **SrcDescList** is used to determine the logical objects involved, and the data management system supplies the physical device allocation.
2. Determines the control ports that will be used to communicate set up control information for each part of the transfer, and determines the data port protocol options. Assume control ports are allocated to maximize local transfer rates.
3. Maps the data as stored on physical media to the control port which will be used to control its transfer,
4. Posts listens on each of the control ports,
5. Replies to the client with the control port addresses and their data assignments. Each control port may reply to client independently and in parallel.

Server1 is passive and therefore all control ports involved in this transfer return a listen address. Control port addresses (rather than data port addresses) are returned in the **IOR**. Data port addresses will be exchanged later during the port setup phase (see below).

Assume that the data is stripped on three devices (which can send data in parallel), with **50MB** on each device. Server1 may therefore use 3 control ports to maximize the transfer rate locally. Assume these are **Port1a**, **Port1b**, and **Port1c**.

Server1 posts listens on these control ports to wait for Server2 control port connects, and replies to the client with these control port addresses and their data assignments. Each control port may reply to the Client independently and in parallel.

Server1's control ports Reply

```
IOR:{RequestID:     F1F2Move ,
        Status: , Flags:;
        ReplyType:   IOlistenList,
        PortList:{ListLength: 1,
              Listenlist: {
                 Flags:   ControlAddr | XferIP | XferIPI3 | Responder
                 Woffset:00MB, Length:50MB, AnAddress:{AddrType:IP,Port1a}} }
        SrcReplyLength:0,    SrcReplyList:{},
        SinkReplyLength:0,   SinkReplyList:{}                    }
Port1a: receive():
(similarly, Port1b responds with (Woffset:50MB,Length:50,address of Port1b)
and Port1c responds with (Woffset:100MB,Length:50,address of Port1c.
Port1a, Port1b, and Port1c are listening)
```

Each reply to this request contains:

1. The RequestID **F1F2Move** to identify the reply.
2. **Flags** for IOR on return are returned with same as supplied in IOD flags
3. The **ReplyType** is **IOlistenList** to indicate that a control port list is being returned, and
4. The control port list together with their data assignments, and data port protocol options. In this case, control port TCP/IP address of **Port1a** is being returned and this port will be responsible for controlling the delivery of the portion of the overall transfer given by {**Woffset**=0, **Length**=50MB}. The Flags values are:
    1. **ControlAddr** to indicate that **Port1a** is a control port address
    2. The data transfer protocol options **XferIP**, and **XferIPI3** for the server (the data port is capable of using either of these protocols)
    3. **Responder** to indicate 3rd party capable (e.g. IPI-3) devices are involved. Each Server determines if 3rd party capable devices are involved in moving the assigned data {**Woffset**=0, **Length**=50MB}, and if so, this flag is set (when the device capabilities have been determined for a partition on both sides of the transfer, the active server will determine which side will control the data connection). The device will connect and move the data directly to the sink server(s) with the control port serving only to issue commands to the device.
5. This is an intermediate reply **IOR: Src/SinkReplyList**s are null.

2. CLIENT sends IOD to SERVER2.

The Client received the following from each passive port:

Port1a: {Flags:**FromServer1**, WOffset:**00MB,** Length:**50MB**, **Port1a** address}

Port1b: {Flags: **FromServer1**, WOffset:**50MB**, Length:**50MB**, **Port1b** address}

Port1c: {Flags: **FromServer1**, WOffset:**100MB**, Length:**50MB**, **Port1c** address}

An IOD for Server2 is constructed containing a **SinkDescList** of the objects for the active side and a *NEW* **SrcDescList** which has the port assignments of the passive side. The IOD is sent to Server2, and the Client waits for termination status replies (**IOR**s) from both servers: one **IOR** from Server1 and one **IOR** from Server2.

Client IOD for Server2

```
IOD:{RequestID:    F1F2Move,
        Function:      IOwrite,
        Flags: ,
        SrcDescLength:3, SinkDescLength:2,
        ReplyTo: null; always reply to client with final IOR
        SrcDescList:
        1  {Flags:FromServer1,WOffset:00MB, Length:50MB, Next:2,
               AnAddress:{AddrType:IP, Port1a}}
        2  {Flags:FromServer1,WOffset:50MB, Length:50MB, Next:3,
               AnAddress:{AddrType:IP, Port1b}}
        3  {Flags:FromServer1,WOffset:100MB, Length:50MB, Next:null,
               AnAddress:{AddrType:IP, Port1c}}
        SinkDescList:
        1  {Flags: ,WOffset:00MB, Length:75MB, Next:2 ,
            AnAddress:{AddrType:HAN, Handle:File2,HandleOffset:025MB}}
        2  {Flags: ,WOffset:75MB, Length:75MB, Next:null,
            AnAddress:{AddrType:HAN, Handle:File2, HandleOffset:150MB}}        }
```

The IOD for Server2, the active server, contains:

1. The Request Identifier **F1F2Move**
2. The function to perform: **WRITE**
3. The set of flags describing the function,
4. The length of the source and sink descriptor lists,
5. **ReplyTo** is **null;** the final IOR is always sent to client,
6. The source descriptor list containing the control port lists from Server1 as previously described. The Flags associated with each window offset give, high level protocol options, device transfer capability (i.e. whether IPI-3 capable devices are involved) etc.
7. The **SinkDescList** specifies where Server2 is to write the data from the window e.g. the second entry in this list specifies that **Length=75MB** is to be written to file File2 Offset **150MB** and that the data will be coming from **WOffset=75MB** in the overall transfer.

Server2 determines device data layout from the **SinkDescList**, assigns control ports responsible for controlling data movement, and maps physical device data to these ports.

Determine control port interconnect Server2 can now determine the port to port interconnect requirements necessary to transmit the data, and the protocol to use at each port.

Suppose the data is stripped on two devices, **75MB** on Device2a, and **75MB** on Device2b. Suppose that these devices may potentially operate in parallel and that two control ports {**Port2a, Port2b**} are to be used to control data movement and maximize local transfer rates.

Server2 can now determine the port to port interconnect and begin the transfer. When Server2 has determined its local control ports and their data assignments, the passive control port information (**SrcDescList** in this case) is read to determine the interconnect. Since active **Port2a** is responsible for **75MB**, and **Port1a** is responsible for **50MB** at window offset 0, **Port2a** will connect to **Port1a** for **50MB**, and then to **Port1b** for **25MB** etc. The port to port interconnects will in general be the cross product of the ports in the transfer. In this case, the interconnects will be:

Port1a <-> **Port2a**	woffset 0,	length 50MB
Port1b <-> **Port2a**	woffset 050MB,	length 25MB
Port1b <-> **Port2b**	woffset 075MB,	length 25MB
Port1c <-> **Port2b**	woffset 100MB,	length 50MB

The protocol to use on each connection can now be determined by the active server (one which is available to both control ports). The passive Server supplied the protocol options at the port in the IOR.

3. Connect and Data Flow, Case1.

Server1 ports were supplied by Server1's reply to client and were passed to Server2 in its IOD. Server2 uses this to select the protocol. Server2 establishes the protocol option to be used over the connections, and establishes the data connection roles for each side (normally, active side data ports will control the transfer but there is an exception given in Case 2 example below). The protocol options (e.g. IP, IPI-3 in this case) are selected on a port to port basis by Server2 (the active server). Server2 chooses a protocol based on what is available to both sides. Suppose IPI-3 is to be used.

Server2 *(Active)* control ports connect to Server1 *(Passive)* control ports and request data. Passive control ports (which are listening) deliver the data port addresses for the active side to connect to, and the amount of information that can be transferred (the actual data exchange may therefore involve flow control so that the data exchange between data ports will fit available buffers on each side).

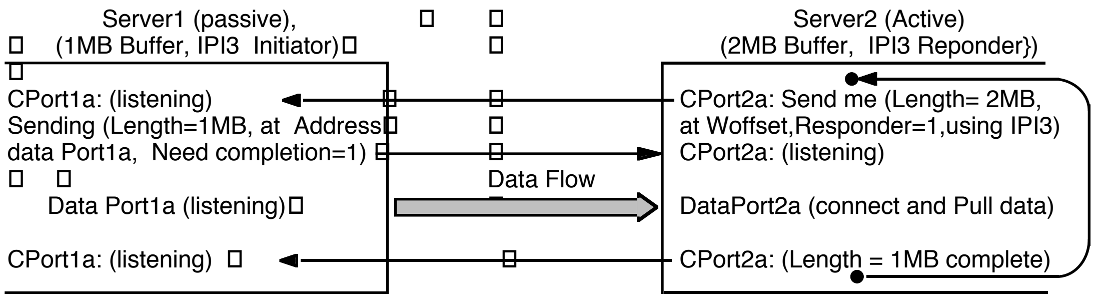
Data ports connect and transmit data: active ports connect and send or request data from passive data ports.

All ports involved in the transfer are connecting, setting up, and transferring data in parallel at this point. The transactions being conducted on one of these connections, between **Port1a** and **Port2a,** would be as follows:

Port1a -> Port2a Server2 creates data ports (if necessary) and connects to Server1 control **Port1a**. **Port1a** is listening for active side connects. Assume Server2 will use Data port**2a**, and Server1 will use Data port**1a** to transfer the data**.**

Suppose a **1MB** buffer is available at Data port**1a**, and a **2MB** buffer is available at Data port**2a.** The entire **50MB** portion of this transfer cannot fit available buffers, and must therefore be further split to fit the available buffers. Flow control is necessary between the port sets.

For each port pair, an *InitiatorMsg* is sent by the active server to start the transfer between the pair: (an *Initiator* makes a request, and a *Responder* drives the connection). The control negotiation and data transport mechanics are performed on a port to port basis as follows:



Server2 reqeuests data at Woffset in the transfer, with Length 2MB (its buffer size), indicates its data port will control the connection (Responder), and the protocol to be used. Server1 replies with its Dataport address (so that the active side can connect), the Length=1MB that it can deliver (its buffer size), and that a completion message is desired. Active side data ports connect and move 1MB (Server1's buffer limit) of data as usual. Server2 returns a completion message to Server1 and then iterates until this 50MB portion of the transfer is completed. The Initiator makes the request, and the Responder controls the connection.

Fig. 13 Flow Control Protocol

Server2 *(active)* builds an **InitiatorMsg** containing the portion of the transfer desired (**Woffset,Length**), the protocol to use, and which side will control the connection during data transport (**Responder=1**). **Delimiter, SecurityTicket, CheckSum,** are loaded as in other messages. **Flags** are **Responder=1** to indicate that the active side data port will drive the data connection.

```
InitiatorMsg{
    {Flags: Responder=1, AddrType: IPI,
     WOffset:0, Length:2MB,TotalLength:50MB}    }
Send(Port1a, InitiatorMsg); Receive();
```

Server2 sends the **InitiatorMsg** to Server1 control **Port1a** (which is listening) and waits for **Port1a** to reply.

Server1 *(passive)* constructs a reply (an **InitiatorMsg)** containing the portion of the transfer it is capable of transferring (**Woffset,Length=1MB**), and the data port address to connect to. **Delimiter, SecurityTicket, CheckSum,** are loaded as in other messages. Server1 creates **Data port1a** if necessary, posts listens, and replies to Server2. **Flags** are **NeedCompletion=1** if a completion message is desired (set by the *Initiator* which is Server1 in this case)**.**

```
InitiatorMsg{
    Flags:NeedCompletion=1,
    WOffset:0, Length:1MB,TotalLength:50MB,
    ConnectTo:{AddrType:IP, addr: Data port1a}    }
Send(Port2a, InitiatorMsg); Receive()
```

<u>Server2 *(active)*</u>. data port **Data port2a** connects and ***Pull***s data from Server1 **Data port1a**, and returns the completion message to Server1 if requested.

```
CompletionMsg { BytesMoved: 1MB };
Send(Port1a, CompletionMsg); Receive();
```

<u>Repeat until data transferred</u>: The Second Iteration would be:

Server2 sends: Offset:**1**, Length:**2MB**,TotalLength:**49MB**
Server1 replies: Offset:**1**, Length:**1MB,** TotalLength: (not used)
Server2 data port: connect and pull **1MB.**

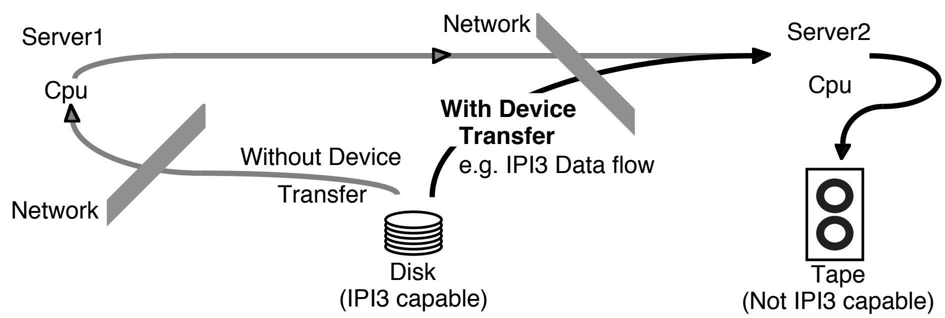
Similarly and in parallel, data is flowing from **Port1b** to **Port2b.**

3. Connect, and Data Flow Case 2.

Third Party (device to device, Active Initiator)

If the devices involved are capable of direct transfer (e.g. use IPI-3, SCSI-3, protocol), then it may be possible to avoid intermediate moves of the data through the servers. The devices may be able to send or receive data directly.

Below, suppose the *Disk* is IPI-3 capable. The control port assigned to move the data from this device will issue the commands to the device, but otherwise serve as a third party in the transfer. The device *Disk* will send the data to, or request the data from, Server2 without passing through Server1.



Above, suppose the Disk is network attached and controlled by Server1 (Server1 knows the data layout on this disk). If a device is capable of controlling a portion of the transfer, then the device can send (or request) the data directly from Server2, rather than moving the data through Server1.

Fig. 14 Intelligent peripherals may reduce intermediate moves

If the IPI-3 capable device is on the active side (or both), then the active side controls the device (e.g. can issue positioning, read, write commands to the device and knows the data layout of the device) and also controls the connection. The device on the active side can send or receive the data directly with no special treatment in this case.

If the IPI-3 capable device is on the passive side however, then the server which controls the device does not control the connection. Direct transfer is not possible unless we let the passive side control this portion of the transfer, because the active side does not know the data layout on the device, how to position the device, or how to issue read or write commands to the device when sending or requesting data.

We therefore exchange active and passive roles for this portion of the transfer, in order to allow direct device transfers even in this case. It is possible to do this in all cases, even when the transfer has sequential ordering requirements, because only a part of an ordered transfer is involved (and therefore the transfer still remains ordered; the active side guarantees the ordering of the overall transfer).

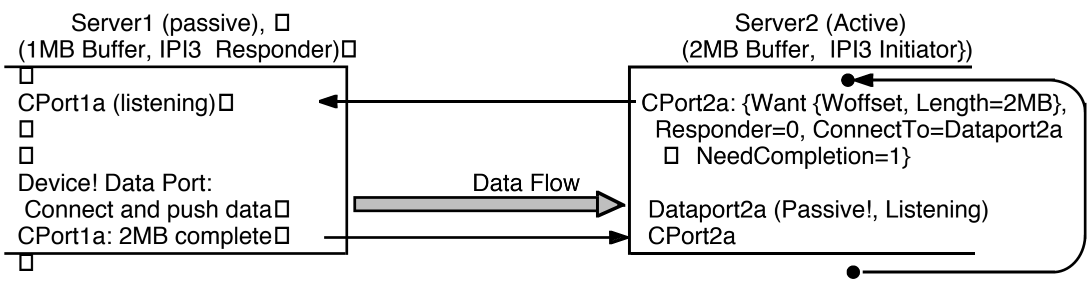
Case	Passive Device	Active Device	Controlling Device
1	IPI-3	IPI-3	Active IPI-3 device controls
2	none	IPI-3	Active IPI-3 device controls
3	IPI-3	none	Passive IPI-3 device controls
4	none	none	Niether: No optimization possible

In cases 1 and 2 above, the active side has an IPI-3 capable device which should control the transfer: the active side controls both the connection and the device. In case 3 however, only the passive side has the IPI-3 capable device and must also be allowed to control the connections.

Fig. 15. Possible Active/Passive Combinations

In Fig. 14, an IPI-3 capable device *Disk* involved in a portion of the transfer, is controlled by the passive side (only Server1 knows the data layout on this disk and only Server1 can issue commands to this device). The active side recipient device is not IPI-3 capable, and furthermore is a tape unit which must be delivered data in the particular sequential order to write the tape. We are still able to allow the IPI-3 capable device to control the connection and push data directly to Server2 for this portion of the transfer.

From the protocol options (**Flags**), the active server, Server2, knows that an IPI-3 capable device is involved in this portion of the transfer on the passive side but not on the active side. Server2 therefore posts data port listens, and indicates to Server1 that it (Server1) will be the IPI-3 *Responder* during the transaction (**Flags: Responder=0**). Server1 will issue commands to the device, but the device data ports will connect and drive the connection (i.e. *Push* data to Server2 to transfer the data). The interaction is as follows:



Server2's Dataport address is passed to the passive side, so that the device can connect and send (or request) data (the device will control the transfer). Note that Server2 **Data ports are listening** even though Server2 is active. Server1 does not need to respond back to Server2 with its port capabilities, buffer size, etc., because its device will be active.

Fig. 16 Flow Control Protocol, with the passive side controlling a connection

Active side (Server2) creates data ports, if necessary, and post listens on these addresses. Server2 data ports are listening for connects.

<u>Repeat until data transferred:</u>

<u>Server2 *(active)*</u>. Server2 builds an **InitiatorMsg** containing the data port address for the device to connect and send data to (**Data port2a**), the transfer descriptor (Woffset,Length, etc), and the protocol options (e.g. IPI-3 in this case). **Delimiter, SecurityTicket, CheckSum,** are loaded just as in other messages. **Flags** are **NeedCompletion=1** if a completion message is

desired, and **Responder**= 0 to indicate that Server2 is the IPI-3 *Initiator*. Server2 connects to Server1 control **Port1a** (if necessary), and sends the **InitiatorMsg:**

```
InitiatorMsg{
    AddrType: IPI, IPI3addr: Dataport2a,
    {Flags:NeedCompletion=1,
    WOffset:0, Length:2MB,TotalLength:50MB}    }
Send(Port1a, InitiatorMsg); receive();
```

Server1 *(passive).* Server1's device (e.g. an IPI-3 device) will control the connection. Server1 Control **Port1a** sends the device 1) a Read request, 2) the address of **Data port1a** and 3) the amount of data to read (**2MB:** there are no Server1 buffering requirements). The device data port **Dataport2a** connects and ***Push***es data directly to Server2 without passing through Server1.

```
CompletionMsg { BytesMoved: 2MB };
Send(Port2a, CompletionMsg); receive();
```

Repeat until data transferred The Second Iteration would be:

Server2 sends: Length:**2MB**, WOffset:**1MB**, TotalLength:**48MB**
Server1: IPI-3 device data port connect and ***Push*** **2MB** to Server2.

Similarly, and in parallel, data is flowing to Server2 **Dataport2b**.

4. Return Completion status and terminate

Each Server collects all completion status from all ports, forms a collective completion status, and replies to the client (which is listening for completion status messages). The Client is listening for one **IOR** from Server1 and one **IOR** from Server2.

Server-1 Replies with Completion status to Client:

```
IOR{RequestID,
    Flags:   IOcomplete,
    Status:          0       (no error; for error definitions, see Errorno.h)
    PortList:        Null,
    SrcReplyLength:3, SinkReplyLength:1
    SrcReplyList             (what we thought we successfully read)
        {Flags:IOcomplete, Status:0, BytesMoved:50MB}
        {Flags:IOcomplete, Status:0, BytesMoved:50MB}
        {Flags:IOcomplete, Status:0, BytesMoved:50MB}
    SinkReplyList            (what we thought we delivered to the other side:
        {Flags:IOcomplete, Status:0, BytesMoved:150MB} }
        (The IOD sent to Server1 did not have a sink list: status is for entire move)
Send(Client, IOR)   client is listening for IO status returns
```

The **Src/SinkReplyList**s mirror the information in the IOD **Src/SinkDescList**s. The client will know how to locate errors in the overall transfer by relating it to the IOD that was sent.

The lists for both sides of the transfer are returned by each side. This allows each side to specify how much information was successfully moved locally versus how much was moved externally. In the above example, the **SrcReplyList** returns how much information Server1 successfully read for each entry in the IOD, and the **SinkReplyList** returns the amount of information Server1 sent to Server2.

Server1's IOD didn't contain a **SinkDescList** (because it was passive during the transfer); the **SinkReplyList** is for the entire move. Server2's IOD can be more detailed.

Server-2 Replies with Completion status to Client:

```
IOR{RequestID,
    Flags:   IOcomplete,
    Status:          0       (no error; for error definitions, see Errorno.h)
    PortList:        Null,
    SrcReplyLength 3, SinkReplyLength 2
    SrcReplyList
        {Flags:IOcomplete, Status:0, BytesMoved:50MB}
        {Flags:IOcomplete, Status:0, BytesMoved:50MB}
        {Flags:IOcomplete, Status:0, BytesMoved:50MB}
    SinkReplyList  (what we thought we delivered to the other side)
        {Flags:IOcomplete, Status:0, BytesMoved:75MB}
        {Flags:IOcomplete, Status:0, BytesMoved:75MB}
Send(Client, IOR)           client is listening for IO status returns
```

The interpretation of **SrcReplyList** in this case would be how much information Server2 believed it received from the Server1 and the interpretation of **SinkReplyList** would be how much information Server2 believed was written.

Error detection:

The Source and Sink reply lists always mirror the original IOD. The Client retains the IOD to locate a given error and relate it to the overall transfer.

Suppose Server1 encounters an error at 100MB (**Port1a** and **Port1b** completed successfully but **Port1c** reported error. **Port1c** reports **BytesMoved=0**. Server1's reply would be:

Server1 detects READ Error at 100MB (Port1C)

```
IOR{RequestID,
        Flags:      IOerror,
        Status:     The Kind of error
        PortList:   Null
        SrcReplyLength:3, SinkReplyLength:1
        SrcReplyList
            {Flags:IOcomplete, Status:0, BytesMoved:50MB}
            {Flags:IOcomplete, Status:0, BytesMoved:50MB}
            {Flags:IOerror, Status: error, BytesMoved: 0MB}
        SinkReplyList
            {Flags:IOcomplete, Status:0, BytesMoved: 100MB }
Send(Client, IOR)           client is listening for IO status returns
```

The Client retains the original IOD to locate the error. Suppose

Server2 detected WRITE Error at 100MB

```
IOR{RequestID,
        Flags:      IOerror,
        Status:     The Kind of error
        SrcReplyLength:3, SinkReplyLength:2,
        SrcReplyList
            {Flags:IOcomplete, Status:0, BytesMoved:50MB}
            {Flags:IOcomplete, Status:0, BytesMoved:50MB}
            {Flags:IOcomplete, Status:0, BytesMoved:50MB}
        SinkReplyList
            {Flags:IOcomplete, Status:0, BytesMoved: 75MB }
            {Flags:IOerror, Status:error, BytesMoved: 25MB }   }
```

Termination The API is responsible error recovery, and any retry procedures. The Client returns the collective completion status to the API.

Definitions and Terminology

Active is the role to be assumed by a server on connection. An active server will establish connections and control connections. Connection based communication is asynchronous. Before any communication can begin, it must be decided which end of the transport connection will assume a controlling role. Contrast **Passive.**

ATM (Asynchronous Transfer Mode**)** A link layer protocol for cell-based switched networks developed for telecommunications that specifies a 48 byte payload and 5 byte header. Bandwidths vary widely depending on the physical transport used (e.g. SONET Europe OC-48 @ 2.4 Gb/s, SONET U.S. OC-3C @ 155Mb/s, Fibre Channel @ 155 Mb/s, T3 @ 44.5Mb/s, T1 @ 1.5Mb/s, others; there is no specific physical layer associated with ATM). ATM is an ISDN (integrated services digital network) combining computing and telecommunications in one integrated package (capable of carrying voice, video and data).

Channel A specialized IO interface between a host and a peripheral device connected only to that host. It is a means of transferring data where all devices that can communicate over the channel with a host are known in advance, the host system retains all the knowledge of the peripheral devices attached to it, any device configuration change requires host software or configuration table changes, and data transfer is usually handled by hardware once an IO operation begins. Examples of traditional channels are HIPPI, IPI-3, and SCSI (*channel* technology has tended to evolve into networking technology).

Client Initiates a request for data transfer service between servers over a control connection.

Connection A virtual communication path established by a transport level protocol.

Control Connection A virtual communication path used to exchange commands and replies among client and servers. Control connections are characterized by small control packet exchanges that can tolerate low bandwidth and are intended for use on low latency networks.

Control Port An address on the network for control information to be exchanged.

Data connection A virtual communication path used to transfer data between servers. High bandwidth networks are intended for data connection use. Data connections are characterized by high bandwidth but typically exhibit high latency.

Data Port An address on the network for data to be exchanged.

Data Management System Data management systems define an abstraction, rules to refer to these abstractions, and a set of operations to manage them. Examples of data management systems are file systems, mass storage systems, data base management systems, and transaction processing systems. The abstraction on which file systems are based, for example, is a 'file', and the precise definition and accessing rules for a 'file' will vary depending on which file system is involved.

Ethernet A 10Mbs (Megabit-per-second) baseband shared media network that uses contention based access.

Fibre Channel A fibre optic link designed for use both as a channel and a (circuit switched) local area network. Transmission speeds range from 12.5MBs, in 12.5MB increments to 100MBs (competitive with HIPPI rates). Multiple protocols are supported over a common physical interface (multiple physical interface types and interconnects of those types), separates the logical protocol from the physical interface.

FDDI (Fiber Distributed Data Interface) A 100-Mbs shared media ring based fiber optic local area network.

File An instance of an object. A logically contiguous sequence of bytes with a name. Files may designate a physical device (e.g. UNIX device files), be allocated to multiple device types, or may be distributed over multiple data management subsystems.

HIPPI (High-Performance Parallel Interconnect) Originally, an 800Mbs channel for connecting local high speed peripherals to a host. Now, a family of standards that also define channel switching devices, flow control at the physical level, and very large packet sizes - features that introduce IO style capabilities to traditional networks. **HIPPI** local area circuit switched networks are now relatively common on high performance computing floors.

Initiator An *initiator* makes a request on a connection; a *responder* drives the connection.

IP (Internet Protocol) An Internet Protocol that supports a collection of networks (the Network layer of the OSI model).

IPI (Intelligent Peripheral Interface) **IPI** is a family of standards which define the communication between hosts and peripheral devices in several layers. The scope of the physical environment can be extended beyond channel applications to network based systems.

IPI-3 (Intelligent Peripheral Interface - Level 3) Defines high-level protocols which specify device independent command sets for various device classes, the packet formats for commands and responses, and a transport protocol to manage communication between **IPI-3** conforming devices. No timing requirements are introduced for data transfer and consequently transmission over long distances is possible. Multi-megabyte transfers are permitted as a single packet (depending on the supporting media).

IPI-3 is in use by a variety of disparate devices types such as disk, tape, communications controllers, and frame buffers. A peripheral may initiate a data transfer, when it is ready to send or receive data, and stream data directly between the communications media and peripheral media without (significant) intermediate buffering. The communicating parties may both be hosts, in which case a transfer can be conducted at the full memory or media speeds involved.

Network refers to an IO interface that provides a means for any pair of hosts or devices connected to the network to communicate with each other at any time. Networks are unstructured and unpredictable environments which require substantial software involvement in the information flow for verifying access permissions, setting up sessions, and routing transactions to the correct software service. *Network* and *Protocol* are used interchangeably. Examples of traditional networks are IP, ATM, IEEE 802. Networks usually support any host-to-host communication currently rather than host-to-device combinations. Parallel Transport anticipates the increasing addition of intelligent peripherals to the mix of devices on the network and provides the appropriate machinery.

Network Attached Devices Network attached devices are intelligent peripherals which are directly connected to a network and capable of moving data (sending or receiving) directly over physical network connections in response to remote client commands. A client may share control, with other clients or servers of such devices.

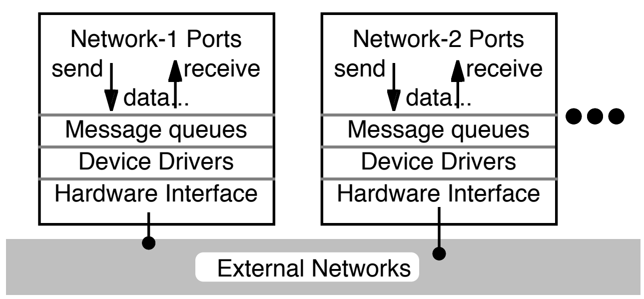
Objects An addressable logically contiguous sequence of bytes. Objects may be files, devices, network locations, memory buffers, data base references, or other addressable abstractions that may serve as the source or destination of data. The addressing rules locating an *object* will be defined by, and vary, dependent on which data management systems are involved in a transfer. In a UNIX file system, for example, a path name or handle locates an object.

OSI reference model. The International Standards Organization (ISO) seven-layer Open Systems Interconnection (OSI) protocol model for communication over a network.

Passive The role one end of a transport connection assumes for the purposes of establishing and controlling a communication. A passive port will listen for connect requests and subsequently commands from an active port.

Port A network address for two or more processes to establish a connection. A queue of messages with a network address.

Ports are a common abstraction supplied by operating systems which provide a rendezvous point for processes to communicate. Ports manage a queue of messages typically. The communicating processes may be local or external to the server. External ports (protocol ports) are associated with a specific protocol (network). Operations open, send, receive, and close are defined on ports. A generic view of an external interface is:



Protocol A standardized set of rules used for dialogue between a sending and receiving host.

Responder An *initiator* makes a request, while a *responder* drives a connection.

Server A server supplies (a data movement) service to a client. A Server stores data and moves data over data connections in response to Client control requests.

Sockets A session layer network programming interface which provides for opening a session (establishing addresses and connections), sending and receiving data, and closing a connection.

TCP (Transmission Control Protocol). An Internet transport protocol (OSI layer 4) which provides *reliable delivery* of a byte stream. Packetization, end-to-end flow control, error detection, recovery, and possibly some congestion control is performed by TCP. Data is received from the Session layer (e.g. Socket send) or Network layer (e.g. from IP as a result of a Socket receive call)

References

[DDN85] 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.

[DCE93] Open Software Foundation, 11 Cambridge Center, Cambridge MA, Distributed Computing Environment Documentation set, Version 1.0.3, 5/93. Open Software Foundation.

[SSS93] IEEE Storage Systems Standards Working Group (Project Authorization Request 1244). 'Reference Model for Open Storage Systems Interconnection'. Mass Storage Systems Reference Model Version 5. Draft in progress 8/93.

[Cor93a] Peter F. Corbett, Sandra Johnson Baylor, Dror G. Feitelson, 'Overview of the Vesta Parallel File System', Proceedings of the Workshop on I/O in Parallel Computer Systems, International Parallel Processing Symposium, April 13, 1993 Newport Beach CA.

[Cor93b] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, Sandra Johnson Baylor, Parallel Access to Files in the Vesta File System, Pg. 472, Proceeding of Supercomputing '93, IEEE Computer Society Press 11/93.

[Coy93] Robert A Coyne and Richard Watson, 'Storage Systems for National Information Assets', Proceedings Supercomputing 1992', Pg. 626, IEEE Computer Society Press.

[CHW93] Robert A. Coyne, Harry Hulen and Richard Watson, 'The High Performance Storage System', Proceedings Supercomputing 1993.

[Cro89] Thomas W. Crockett, 'File Concepts for Parallel IO', Proceedings of Supercomputing 1989', Pg. 574, IEEE Computer Society Press.

[EPD93] Erik P. DeBenedictis and Stephen C. Johnson, 'Extending Unix for Scalable Computing', IEEE Computer 11/93.

[EPD91] Erik DeBenedictis, P. Madams, 'nCUBE's Parallel IO with UNIX Compatibility', Proceedings of the Sixth Distributed Memory Computing Conference', April 1991, Pg. 270.

[Kot92] David Kotz, 'Multiprocessor File System Interfaces', Proceedings 1992 USENIX Workshop on File Systems, 5/92.

[RHW93] Rich Ruef, Robert Hyer, and Richard Watson, 'High-Performance data Transfer using Network-Attached Peripherals at the National Storage Laboratory', Proceedings of the Twelfth IEEE Symposium on Mass Storage, IEEE Computer Society Press, 4/93.

[TMC93] Susan LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, Richard Wheeler, 'A Parallel File System for the CM-5', Proceedings of the 1993 Summer USENIX Conference, Pg. 291.