SPIRAL: A Client-Transparent Third-Party Transfer Scheme for Network Attached Disks

Xiaonan Ma
Texas A & M University
xiaonan@ee.tamu.edu

A. L. Narasimha Reddy
Texas A & M University
reddy@ee.tamu.edu

Abstract

Third-party transfer is a data transfer mechanism where the party initiating the transfer is neither the source nor the sink for the data. In this paper, we present a scheme for supporting third-party transfers on storage systems with network-attached disks (NADs), called SPIRAL [1]. SPIRAL allows NADs to send data directly to clients without going through the server. It is transparent to clients and relies only on the linear block interface of the current disks (or the future iSCSI disks). SPIRAL requires no porting of file-system or application-level functionality to NADs and requires only simple modifications to server applications. To illustrate our approach, we implemented a prototype system on PCs running Linux. We present experimental results for NFS and HTTP on the prototype to demonstrate the effectiveness of our approach.

1. Introduction

Storage devices today are increasingly interconnected to servers through switched networks instead of buses. Fibre channel systems and recent IETF efforts in IP storage (e.g., iSCSI) are significant industry milestones in this direction. It is expected that these network based storage systems can leverage research and development efforts in networking technology to improve the device and system interconnection speeds. Further, IP based storage systems are expected to consolidate the network wiring infrastructure of data centers.

Recent advances in compression and communication technologies have resulted in the proliferation of applications that involve storing and retrieving multimedia data such as images, audio and video across the network. With increased disk capacities and network bandwidth, larger multimedia files are being stored and retrieved from networked servers. Compared to traditional file access, these multimedia files pose different challenges. First, large volumes of data need to be handled by the server's memory and IO subsystems. Traditional client-server protocol stacks and OS software layers often require data to be copied several times for them to be transfered across the network. Second, large data transfers reduce the effectiveness of the server's buffer cache system, increasing the number of page faults for other data such as file system metadata and smaller files. These problems get compounded by the extra network to memory transfers and extra network stack processing when storage devices are attached to the network. In this paper, we will focus on storage systems with network attached disks (NADs).

It has been shown that third-party transfer can significantly improve the scalability of storage systems [8, 18]. Third-party transfer is a data transfer mechanism where the party initiating the transfer is neither the source nor the sink for the data. For storage systems with NADs, it normally refers to the ability of the system that allows direct data movement between the NADs and the clients through network. Third-party transfer reduces the amount of data traffic through the server and improves the scalability of the system. However, existing approaches normally require the clients to be modified to communicate with NADs directly, or require porting file-system and application-level functionality onto the NADs. Client-side modification may be acceptable for a small closed environment, but is impractical for public web servers and video servers accessed by large number of clients scattered over wide areas. It also requires significant amount of development and maintenance efforts if the clients are running on different platforms. Porting file systems or applications (such as NFS or HTTP server) to NADs could be very difficult if the NADs are based on a block-based interface, for instance, iSCSI.

In this paper, we propose SPIRAL, a scheme for supporting third-party transfer on storage systems with NADs. SPIRAL enables NADs to send data to clients directly over the network (LAN or WAN) instead of through the server. It has the following characteristics:

[1] SPIRAL stands for "Server-side Packet Interception and Redirection At Link level".

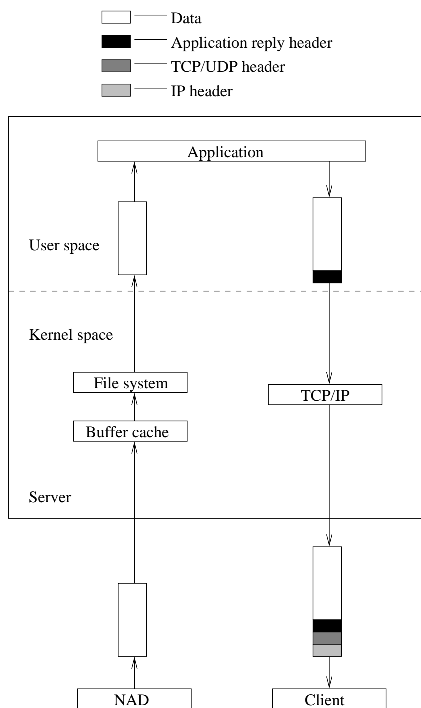

Figure 1. Data flow for read in traditional systems

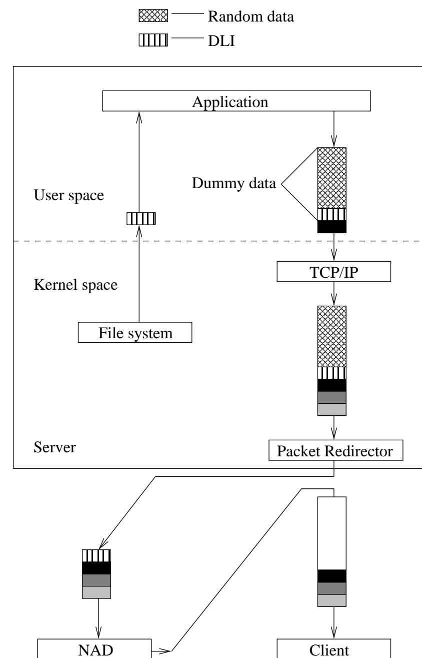

Figure 2. Data flow for read in SPIRAL

- It is **client-transparent**. There is no need to modify the OS and applications on the client side.
- It uses existing **block device interface**, allowing simple reuse of existing file system and operating system technology.
- It keeps the software layer on the NADs as thin as possible for efficient utilization of device resources. No file-system or application-level support is required on NADs.
- It supports both UDP and TCP.

SPIRAL's approach is aligned with recent IETF's standardization of iSCSI which allows many interesting scenarios of detaching block-level storage services from file servers in the future. For example, storage-as-service business models could employ SPIRAL to allow storage devices to be pooled in one area while retaining the file servers at customer premises. SPIRAL also solves the problem of supporting third-party transfers from multiple NADs over a single TCP connection (as required for NFS over TCP or persistent HTTP).

It should be noted that currently SPIRAL only supports third-party transfer for outgoing data from server to clients. Also, data moving operations may only contribute to a fraction of the total workload on a server, in which case the performance gained by directly moving the data between NADs and clients could be limited [8]. Therefore, SPIRAL is best suited for workloads where a large part of the server's load comes from transferring data to clients, such as the case for networked video servers.

The rest of the paper is organized as follows. Section 2 presents the design and the key ideas of SPIRAL. In Section 3, we describe some details about our prototype implementation and discuss the results of running NFS and HTTP on the prototype. Section 4 discusses related work and Section 5 concludes the paper.

2. Design

Applying third-party transfer for all data requests will cause the system to lose the benefit of caching at the server. In SPIRAL, it is expected that the server serves metadata requests and small files directly to clients, while utilizing third-party transfers for large data transfers. Such an approach combines the strengths of distributing server load for large data transfers and centralized caching for metadata and small files. The server can decide when to em-

ploy third-party transfers based on request type, size of data to be transfered, or a combination of several criteria. Alternatively, third-party transfer can be specified explicitly through such schemes as Active Names [24] and other namespace based approaches. In practice, which scheme should be used depends on the actual workload, application and system configuration.

We assume that NADs support a linear block level interface as the current disks. To keep the software layer as thin as possible on NADs, SPIRAL retains all the file system functionality and applications at the server. Most of the system level modifications are kept on the server. We also assume that NADs can communicate with the server in a secure way, either through the use of a private network or through cryptographic mechanisms.

SPIRAL is based on two key ideas. First, SPIRAL introduces a new file system interface `tp_read`. Server applications that support third-party transfers call `tp_read` instead of normal read for those data they want to be directly sent back to clients by NADs. Similar to normal read, `tp_read` contains codes for checking the validity of the parameters, whether the requested data are locked, etc. The main difference between `tp_read` and normal read is that: instead of fetching the data to the server's buffer cache and then copying them to the application's user space buffer (also called user buffer in the rest of the paper), `tp_read` only resolves the location of the data and writes **data location information (DLI)** in the front of the supplied user buffer. DLI normally contains the disk ID, block numbers of the blocks containing the requested data, start and end byte offset, etc. The start and end byte offset indicate the start and end position of the requested data in the first and the last data block, respectively. The size of a DLI is typically a few dozens of bytes and is usually much smaller than the size of the requested data. The rest of the user buffer remain untouched. At the end of the call, `tp_read` returns the correct number of bytes to the caller as if the read has been done in the normal way (in the rest of this paper, we refer to what is inside the user buffer after `tp_read` returns as "dummy data"). The application then "pretends" that it has already read the real data from the file and sends out replies using the dummy data to the client. Here we assume that the difference between the dummy data and the real data does not affect the correctness of other fields in the reply. We will discuss more about this assumption in section 2.6.

Second, SPIRAL uses a datalink-layer selective **packet redirector** as shown in figure 2. This redirector sits on the server and monitors all outgoing packets to intercept packets that contain dummy data. Since dummy data are not real data, these packets should not be received by the clients as they are. The redirector shrinks these packets by removing all dummy data (except for the DLIs) from the payload of the packets, retaining all other information such as the IP headers, UDP/TCP headers and the application reply headers. These shrunk packets are then redirected to the NADs. Note that unlike in Slice [4] where packets are redirected by manipulating packet headers, shrunk packets in SPIRAL are actually tunneled to the NADs with the packet headers treated as normal payload. Tunneling the packets simplifies the receiving code on the NADs and allows us to leverage the existing security protection scheme of the communication channels between the server and the NADs. We will use the term "redirect" and "tunnel" interchangeably in this paper. After a NAD receives a redirected packet, it inflates the packet to its original size by inserting data read based on the information in the corresponding DLI. The NAD then sends this packet out to the clients through a raw socket after recalculating the TCP/UDP checksums. There is no need to update the IP checksums since the IP headers remain untouched. Because the packet now looks exactly the same as if it was sent out by a normal server directly, there is no need for any modification on the client.

Figure 1 shows the data flow from a traditional data server with NADs to clients. Clients send data requests to the server, the server fetches the data from the NAD to its buffer cache and then sends the replies to the clients. Typically, there are several data copies associated with this procedure: data needs to be first copied into the server's buffer caches, then copied to the user space buffer of the application. Another data copy is required for the data to go through the kernel network protocol stacks before they are sent out to the client. Figure 2 shows the typical data flow for third-party transfers in SPIRAL. Except for the DLIs, the rest of the dummy data are acting as place holders and do not contain valid information. In the figure, we denote those data as random data though in practice they may not be truly random. From the figure we see that: (1) data transfers from the NAD to the server memory are avoided; (2) the server's network stack buffers are released sooner since dummy data are no longer needed after the packet is processed by the packet redirector. In SPIRAL, metadata requests and small file transfers still follow the traditional data flow shown in figure 1.

One detail left out in figure 2 is that the packet redirector may add a little more information such as the offsets of dummy data in each packet when redirecting packets. Such information is not available when `tp_read` composes DLIs but is necessary for NADs to insert data in the right position. Figure 3 gives a closer look at how a shrunk packet is tunneled and also shows the structure of a typical DLI. The "dummy data offset info" points to the DLI section of the redirected packet, which will be replaced with the actual data at the NAD.

The advantage of intercepting packets at datalink layer is that all the redirected packets have complete IP and

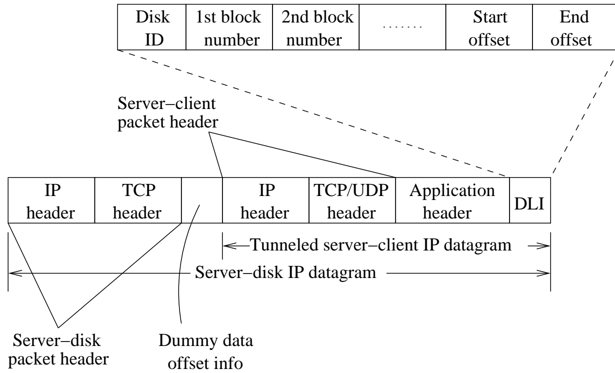

Figure 3. Typical structure of a redirected packet

TCP/UDP header information, so there is no need to maintain transport-level state information for connections with clients on the NAD side. All network issues such as acknowledgements and retransmissions are handled by the server. Since these packets also contain application-level reply headers, the NADs do not have to support individual server applications either. The complete state information, both application-level and transport-level, is maintained only at the server. This allows the server to offload only data transfers to NADs while performing all application-level processing. As a result, the NADs only need to perform operations which are application-independent. Instead of intercepting packets at datalink layer, an alternative scheme may choose to handoff the connections to NADs for data transfers using a TCP handoff protocol similar to that proposed in LARD [20]. For such an approach, frequent connection handoffs may be necessary when data are striped across several NADs or multiple requests are served through one connection (e.g., NFS over TCP, persistent HTTP), incurring significant overhead. We will discuss more about this in section 2.2.

In SPIRAL, all dummy data still need to be copied from user space to kernel network buffers even though most of the dummy data will not be used. This is done to maintain the correct transport-layer state information in the network protocol stacks at the server. The load on the server can be further reduced if this data copy can be eliminated. It is possible to modify the applications on the server so that they can directly compose and send out redirection requests to the NADs, instead of sending replies containing dummy data to the clients. These redirection requests will look similar to the redirected packets in SPIRAL. That is, they will contain DLIs and other necessary informations such as network and applications level headers. However, this approach requires more modifications to the server applications and will only work with connection-less protocols, such as UDP. For TCP, it requires either a TCP handoff protocol, or substantial modifications to the network protocol stacks for allowing the connection state information to be updated even though the data will not be sent out by the server.

As we will show later through experiments, SPIRAL offers several benefits compared with a traditional system. It greatly reduces the amount of data traffic between the server and the NADs. The server only needs to read the metadata for building the DLIs. The bandwidth required for transferring the metadata is normally only a tiny fraction of that when third-party transfer is not supported. This can improve the system's performance significantly when the interconnect bandwidth between the server and the NADs is the bottleneck. There are much less data to be received by the server from the NADs. Less data going through the network protocol stacks helps to reduce the CPU load on the server. Since the data no longer go through the server's buffer cache, the memory usage on the server is reduced too. This will decrease the number of page faults on the server and increase the cache hit rates for other requests.

Though the basic idea of our approach is straightforward, there are several challenges and complications. For instance, how should we implement the packet redirector while minimizing the impact on the performance of other network activities? How do we distinguish, at the datalink layer, the packets to be redirected to the NADs? How do we handle network issues such as IP fragmentation and TCP retransmission? How do we ensure file system integrity and consistency? How to deal with packets that contain multiple pieces of dummy data for different NADs? What about security? In the rest of this section we will discuss each of these issues in detail.

2.1. Packet filtering

To minimize the impact on other network traffic, SPIRAL uses a two level filtering scheme in its packet redirector. In the first level, packets are filtered based on their UDP/TCP port numbers in a way similar to that of a firewall. Outgoing packets of applications not supporting third-party transfer are filtered out at this level. In the second level, packets are filtered based on information provided by the applications. Outgoing packets that belong to applications supporting third-party transfer, but do not contain dummy data are filtered out.

2.1.1. Filtering based on port number. Port numbers for each application that supports third-party transfer can be registered in the packet redirector during system startup or application initialization. Applications can also register port numbers during run time dynamically. Such filtering

requires little processing so the impact on network traffic of other applications is negligible.

One problem with filtering based on port numbers is how to deal with IP fragments since the port number information is only contained in the first fragment. The result is that all outgoing fragmented packets need to be intercepted and grouped based on their fragmentation IDs. Each fragment is then filtered based on the port number in the corresponding first IP fragment. Our prototype implementation shows that the overhead of doing this is very small. Also, many TCP implementations these days use Path MTU discovery to avoid IP fragmentation.

Though the discussion here focuses on port numbers, filtering at this level can be easily generalized to let applications to specify filtering rules that include IP address ranges or other fields in the packet header.

2.1.2. Filtering based on application information. Not all packets sent out by a server application supporting third-party transfer need to be redirected. For instance, reply packets for metadata requests or small file requests should be delivered to clients directly.

In SPIRAL, the server application is modified to provide the packet redirector information for deciding which packets should be redirected. Depending on the application, there are different ways of doing it. The first way is to let the application notify the packet redirector which replies contain dummy data. For example, an NFS [22] server can notify the packet redirector that a particular NFS reply contains dummy data by specifying the reply's RPC XID and client IP address (the XID is guaranteed to be unique for each NFS request from one client). The second way is to let the application provide the packet redirector with low level information such as byte ranges. For example, an HTTP server can inform the packet redirector that the 8000th to 12000th bytes of data (counted from the start of the stream) that will be sent out from a particular TCP connection contain dummy payload. The former requires little change to the application but requires the packet redirector to do some application-level parsing of the packets to track the boundary of each reply and to locate DLIs. The latter simplifies the task of the packet redirector but may require more modifications to the application to keep track of how many data have been sent out for each connection.

In practice, which scheme should be used depends on how easy it is to track the reply boundaries and to parse the replies at datalink level. While UDP maintains message boundaries, TCP provides a byte-stream service and does not lend itself to easy identification of logical reply boundaries. However, applications and higher level protocols may provide their own framing support. RPC over TCP, for example, uses a simple record marking scheme that allows easy boundary tracking at the network level. On the other hand, HTTP does not have such a simple framing support for persistent connections and makes it more difficult for the packet redirector to track and parse the replies correctly. Similar issues on application-level framing have recently been discussed in the context of RDMA support for iSCSI devices [9].

2.2. TCP related issues

As we mentioned previously, alternative approach for the server to offload data transfer to the NADs is to use TCP handoff. For NFS over TCP, large file transfer is broken into a series of fixed size data requests. Using TCP handoff may require the connection to be handed back and forth for each of these NFS read requests, otherwise the NAD needs to be able to build NFS reply packets by itself. For HTTP, though data transfer typically will not be split into small requests, similar problem may still happen when the requested data are striped over multiple NADs. In this case, the server may need to frequently handoff the same connection to different NADs.

Redirecting packets at datalink level also simplifies the disk-side processing, allows NADs to support implementation-specific TCP options between server and clients directly in most cases even if they are not implemented on the NADs. However, these advantages do not come without cost. There are several complications for supporting applications over TCP in SPIRAL.

2.2.1. TCP Retransmission. When the server detects a packet loss in a TCP connection, it will retransmit the packet using data from the TCP send buffer, which may contain dummy data. In this case, the retransmitted packet also needs to be redirected. However, the retransmitted packet may only contain a partial segment of a reply, and may not contain the corresponding DLI. To solve this problem, the packet redirector remembers previous redirections containing information that may be used in potential retransmissions It does this for each TCP connection it is monitoring. Every time a TCP packet is redirected, the start and end sequence numbers along with the corresponding DLI for each piece of dummy payload in the packet are saved in a list for that TCP connection. The packet redirector detects retransmission by comparing the sequence number range of the current packet with the maximum sequence number it has seen for that connection. Since the packet redirector resides on the server, it sees the packets in the same order as the network layer sends them. By comparing the sequence number range of the retransmitted packet with the saved entries in the list associated with that connection, we can find out whether it contains dummy data. A saved entry can be released after all the data specified in it have been acknowledged by the client. Since normally the

length of each list is fairly short and the size of each saved entry is quite small, maintaining these lists consumes little CPU and memory resources on the server.

2.2.2. Inbound filtering. So far, we only discussed about filtering outgoing packets on the server. Inbound packet filtering is also needed for TCP in SPIRAL for mainly two purposes: 1) to track how much data have already been received by the client in order to free the saved entries in the packet redirector when they are no longer needed; 2) to detect termination of a connection for releasing other state information maintained by SPIRAL for that connection. With both inbound and outbound filtering, the packet redirector can detect connection terminations when it sees FIN packets from both sides and all the data have been acknowledged. It will also notice if the connection is aborted by TCP RST packets from either direction. However, a TCP connection may also be aborted by ICMP messages, which will not be intercepted by our packet redirector. For such cases, a garbage collector can be used to release state information in the packet redirector for those TCP connections that no longer exist.

2.3. File system integrity and consistency

It is possible that when `tp_read` is called, part of the data that the client requests are already in the server's buffer cache and some of these blocks may be dirty. The `tp_read` function could first flush the dirty blocks to the NAD before writing the DLI into the user buffer. A more efficient (also slightly more complicated) approach is for `tp_read` to fill the user buffer partially with those data in the buffer cache and apply third-party transfer for the rest.

This does not solve the problem completely though. Since `tp_read` does not read file data but only resolves DLIs, problems arise due to the delay between when `tp_read` resolves the DLI and when the NAD reads the data and inflates the redirected packets. For example, during this period, another process may remove or truncate the file, causing all blocks of that file to be released. Some of the released blocks may be reallocated to another file and overwritten with new data. Part of these blocks may be flushed to the NAD. All these could happen before the redirected packets arrive at the NAD. In this case, the NAD will read the wrong data and send them to the client based on the obsolete information in the DLI.

SPIRAL solves this problem by letting `tp_read` send a short notification message to the NAD before returning to the caller. The notification message contains the same block numbers as those in the DLI. It is important that `tp_read` makes sure that no other writes to these data blocks could arrive at the NAD unnoticed before the notification message. If messages from the server to the NAD are delivered in a FIFO order, `tp_read` can return to the caller after sending out the notification message. Otherwise, it needs to wait until the acknowledgement for the notification message is received from the NAD. When the NAD receives the notification message, it marks these blocks as "pending", which means that the NAD is likely to see redirected packets with DLIs referring these blocks in the near future. If write requests arrive for a pending block, the current content of the block is saved before the block is overwritten. It is possible that the same block number contained in two different DLIs refers to different data, for instance, if a write to that block happens between the two `tp_read` calls. To distinguish them, each notification message contains a unique sequence number which is also carried in the corresponding DLI. The combination of block number and sequence number allows NADs in SPIRAL to identify different versions of a pending block.

Because the same version of pending block may also be referred in several DLIs at the same time, reference counters are used. A particular version of a pending block is released when its reference counter drops to zero. The releasing of the pending blocks is further complicated by retransmission. For TCP, we can not simply decrease the reference counter of a pending block when the NAD finishes processing a redirected packet containing that block, since a retransmission may happen later that requires the same data again. Our solution is to have the server explicitly issue messages to the NAD to decrease the reference counters of the pending blocks. A release message can be sent by the server if we know that the corresponding DLI will not be referred later. This happens when the server receives ACKs from the client that acknowledge all the data specified by the DLI. The approach is similar to release consistency [15]. In practice, the release messages can be sent out to the NADs in batches for efficiency reasons. For UDP-based NFS, the reference count on the pending blocks can be reduced after the corresponding redirected packets are sent. If one or some of these UDP packets are lost, NFS/RPC will re-send the request which will cause a new notification message to be issued by the server.

Note that applications should not cache DLIs and reuse them since doing so provides little benefit but complicates the releasing of pending blocks. Fortunately applications such as NFS and HTTP server normally do not cache read results since the data are cached by the system buffer cache.

For comparison, figure 4 shows the detailed procedure for one third-party transfer in SPIRAL and figure 5 shows the procedure for one regular transfer in a traditional system. In the figures, thick line indicates where bulk data copy/transfer occurs.

Using notification messages also helps to reduce the latency since the NAD can start reading the data blocks into memory without having to wait for the redirected packets.

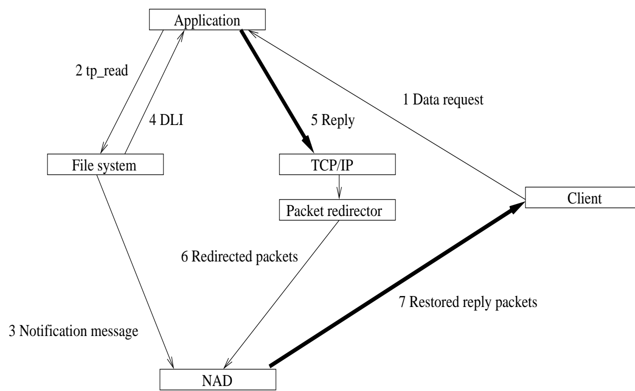

Figure 4. Third-party transfer procedure in SPIRAL

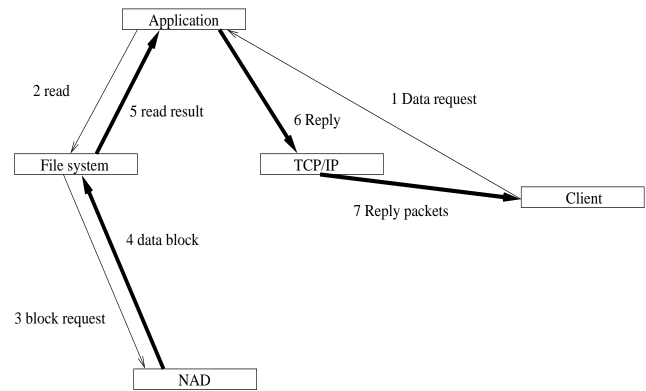

Figure 5. Data transfer procedure in a traditional system

2.4. Multi-disk redirection

Sometimes a single packet may contain multiple pieces of dummy data whose corresponding file data reside on different NADs. This could happen, for instance, when several files from different NADs are required through a persistent HTTP connection, or when data in a single file are striped over several NADs. There are several approaches to deal with such packets. The simplest may be just to let all the NADs involved to circulate the packet and fill in each piece of data in order. The complete packet is sent to the client by the last NAD. Alternatively, the host could split the packet into several IP fragments and send one to each of the NADs. In such a case, the packets are processed on each NAD independently and no data need to be transfered between the NADs. The packets will be pieced together at the client at the network layer. Since the stripe size and the size of the files in third-party transfers are normally much larger than SMSS, the number of these packets should be small compared with the total number of packets that are redirected.

A Volume Manager (VM) adds another level of indirection between file system and disk data layout. Without VM, DLI resolution can be simply implemented through the file system's `bmap` interface. With VM, the information returned by `bmap` may no longer be enough. If the VM is implemented at the disk side (in the NAD controller), then using `bmap` to resolve DLI is sufficient since the volume to device mapping will be done on the NADs. If the VM is on the server, it may need to be modified to provide support for DLI resolution in SPIRAL.

2.5. Security

Since SPIRAL only redirects outgoing data and all the application-level processing is still handled by the server, there is no requirement for NADs to perform access control tasks (such as file permission checking and client authentication). Also, because the NADs do not accept any requests from the clients, the security of the system is not affected. We assume that the server can communicate with the NADs securely. This is not a limitation of SPIRAL because such secure communication channels between the server and the NADs are required for any storage system with NADs, whether SPIRAL is used or not. In SPIRAL, the redirected packets are tunneled to NADs as normal data payload, therefore no extra protection scheme is required.

2.6. Limitations

Currently SPIRAL only supports third-party transfer for outgoing data from the server to the clients. For incoming data from clients, if the data are redirected at the server, then there is no reduction of network traffic. If the data need to be redirected at the router, then the router needs to perform application-level processing. Redirecting incoming data to NADs directly also requires the NADs to be capable of making space allocation decisions, which we have decided to leave within the file system at the server. Dealing with file system integrity and security also becomes much more difficult.

The third-party transfer scheme in SPIRAL requires the server application to build and send out replies using dummy data, which may restrict its use in some applications. For instance, a streaming video server may want to dynamically change the quality of the video stream according to the connection status, and the application may not be able to build the reply correctly without the real data.

Currently SPIRAL does not support encryption schemes such as IPsec or SSL. It may be possible to let the server share keys with the NADs so that the latter could encrypt the data by themselves. Similar problem will be encountered by other third-party transfer schemes as well.

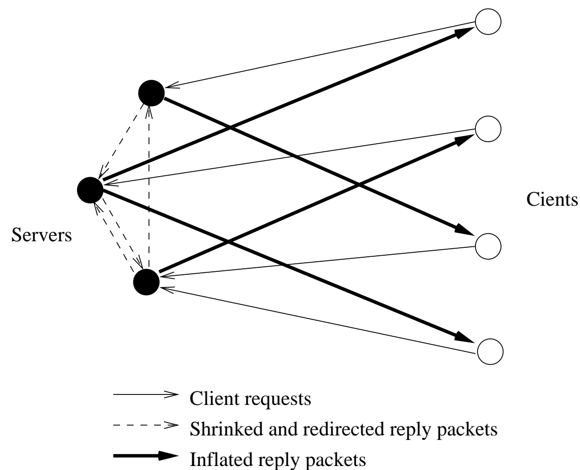

Figure 6. Third-party transfers with SPIRAL in cluster

2.7. Miscellaneous issues

Most file systems perform prefetching to speed up sequential read performance. Though `tp_read` does not fetch data blocks to the server, similar read-ahead strategy can be applied to NADs for prefetching data blocks into on-disk memory. One simple way to implement read-ahead in SPIRAL is to add the block numbers of those blocks to prefetch in the notification messages.

The redirection scheme in SPIRAL may affect the TCP algorithms in the server's TCP implementation through the round-trip time (RTT) values, since the delay caused by the packet redirection will be added to the measured RTT. The disk access latency could also become part of the RTT measured by the server. The result is that the RTT may appear to be longer than what it should be. This may have impact on the congestion control algorithms used in TCP. Since packets containing normal data do not experience this extra delay, this is likely to increase the RTT variance measured on the server. Using read-ahead can help reducing the impact of this since often the data blocks will already be read into the disk's memory when the redirected packet comes.

2.8. Future application

In general, the third-party transfer scheme in SPIRAL should work for NADs supporting non-block interfaces (e.g., an object interface) without significant changes. This can be done by replacing the block numbers in the DLI with object IDs and byte ranges.

Similarly, although our focus is on third-party transfer between the NADs and clients in this paper, there is nothing that prevents the server from redirecting shrunk packets to another server in a cluster. The latter will then fill in the requested data and send the packets to the clients directly. Figure 6 shows a possible deployment of our third-party transfer scheme (with suitable modification) among a cluster of servers. The redirection could happen when the server which holds the connection to the client is not the best candidate for serving the requested data. For instance, it may need to fetch the data from other servers, or another server may have accessed the same data recently and still has the data in memory. Again, for applications which serve multiple data requests over a single TCP connection, such a scheme can reduce the data traffic among the servers without incurring the overhead of frequent connection handoffs.

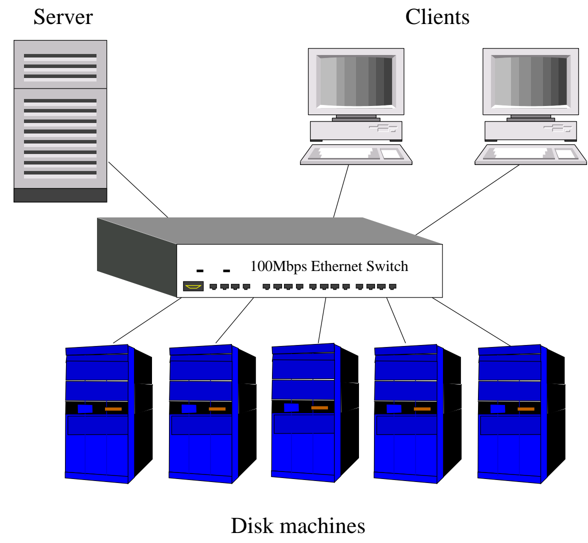

Figure 7. Testbed structure

Besides applications such as NFS/HTTP server, SPIRAL may also be used to improve a system's scalability during remote backup or data mirroring through network.

3. Implementation

The structure of our testbed is shown in figure 7. Our testbed configuration includes a 500MHz PC as the server and a few 166MHz PCs as the NADs. Another several 233MHz PCs act as the clients. The server has 128MB of memory and each client/disk machine has 64MB of memory. Each machine has a 100Mbps network interface card and is connected to each other through a dedicated 100Mbps Ethernet switch. The switch allows each node to send or receive at 100Mbps in full duplex transfer mode simultaneously. The server communicates with the disk machines through the Linux network block device (NBD) driver. NBD allows one machine to use files or disks on another remote machine as its local block devices through TCP/IP. A NBD server runs as a user space daemon on each disk machine. We modified both the NBD driver and the

daemon to add support for SPIRAL. We implemented all components of SPIRAL as Linux loadable kernel modules on the server. Only minor modifications to the kernel NFS server and the Apache web server are required. The details of our implementation are omitted here.

In the prototype implementation, we introduced a third-party transfer file attribute (called TP attribute) through a simple stackable file system layer [11]. The file system layer allows us to add new features on top of existing file system codes conveniently with very little overhead. TP attribute for each individual file can be set or cleared through the `fcntl` file system interface. It allows us to easily turn on/off third-party transfer support in the experiments.

In the prototype, we implemented the port filtering by directly making use of the kernel firewall codes. Our experiments focus on two applications: NFS and HTTP. For NFS, we modified the kernel NFS server's read procedure to first check whether the file's TP attribute is set. If this is true then `tp_read` is called instead of normal read. After `tp_read` returns, the server marks the reply in the packet redirector using the RPC XID and client's IP address before sending the reply out. The modification only adds a few dozen lines of C code. For application-level filtering, a simple RPC parser is developed to check each NFS reply's XID to decide whether the reply contains dummy data. We use a shallow finite state machine to keep track of RPC record (one RPC message fits into one record) boundaries for NFS over TCP. Packets requiring redirection will then go through a simple NFS reply parser to locate the dummy data in the packet. Since only read replies will be signaled as containing dummy data by the NFS server, the NFS reply parser only need to know about NFS read reply format. For HTTP, we modified the Apache web server so that it provides the packet redirector byte ranges of dummy data in the data stream. The modification turned out to be quite straight-forward since the Apache server already contains codes for keeping track of how many bytes have been sent for each reply body. Since this approach does not require the packet redirector to know about HTTP, application-level filtering is done simply based on the byte ranges.

We implemented all components of SPIRAL as Linux loadable kernel modules on the server, including the RPC and NFS parsers, byte range filter and the redirection module that shrinks and redirects packets.

In the following experiments, we compare the results of NFS/HTTP with and without SPIRAL. The absolute performance numbers in our measurements are low by today's standards due to the age of the equipment used, but the relative numbers and conclusions remain valid.

3.1. Large requests

3.1.1. NFS results. In this test, one file system is mounted on the server from each disk machine through NBD. All the file systems are exported to the clients through the modified kernel NFS server. We initiated a client thread requesting a 32MB file on the server sequentially for each disk machine in the system. Figure 8 and figure 9 show the total throughput of the NFS server over UDP and TCP respectively with different number of NADs. The results of SPIRAL are compared with those of an unmodified NFS server without third-party transfer support. The results show that the throughput of the system when third-party transfers are enabled (NFS + SPIRAL + TP) scales almost linearly as the number of disk machines increases. The throughput for normal NFS (NFS) is saturated at around 6MB/s as the server's CPU became the bottleneck.

We also show in the figure the results of SPIRAL when the TP attributes for the files are not set (NFS + SPIRAL - TP). This is done to measure the overhead of SPIRAL to other data traffic where third-party transfers are not utilized. The overhead here includes those from port filtering, IP fragments handling (for NFS over UDP) and RPC parsing. The results show that the performance impact is negligible. The overhead for data traffic of other applications will be even smaller since application-level filtering (such as that performed by the RPC parser) will not be needed.

3.1.2. HTTP results. We did similar experiments for HTTP by replacing the kernel NFS server with the Apache web server and letting the clients retrieve the same files through HTTP. The results are shown in figure 10. Again, the results show that SPIRAL allows improved scalability with the number of NADs (HTTP + SPIRAL + TP) and the overhead on regular traffic is negligible (HTTP + SPIRAL - TP). We also note that with similar configurations, the HTTP results are a little better than the corresponding NFS over TCP results, even though the Apache server is running in user space while the NFS server is implemented using kernel threads. This is because NFS over TCP has more overhead for large file transfers compared with HTTP: there is only one HTTP request from the client for the whole file but many NFS requests. The HTTP server performs authentication once per file compared to authentication per read request in NFS. Moreover, the NFS server has to perform processing such as XDR coding and decoding for each request.

To show that SPIRAL can greatly reduce the requirements on the interconnect bandwidth for the server, we carried out another HTTP experiment where the network bandwidth of the server and the disk machines is limited to 2.5MB/s using the Linux network QoS support [3]. Fig-

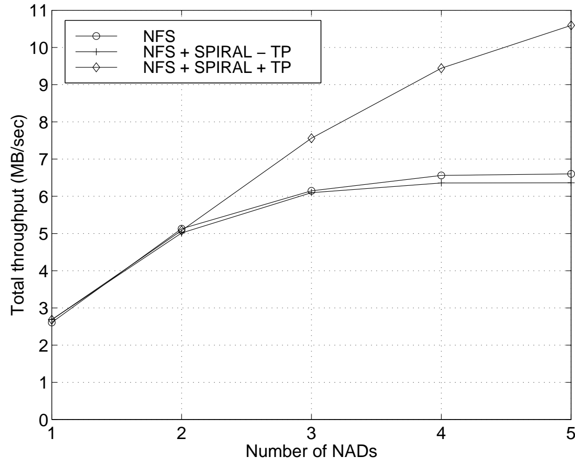

Figure 8. Throughput of NFS (UDP) for large file transfers

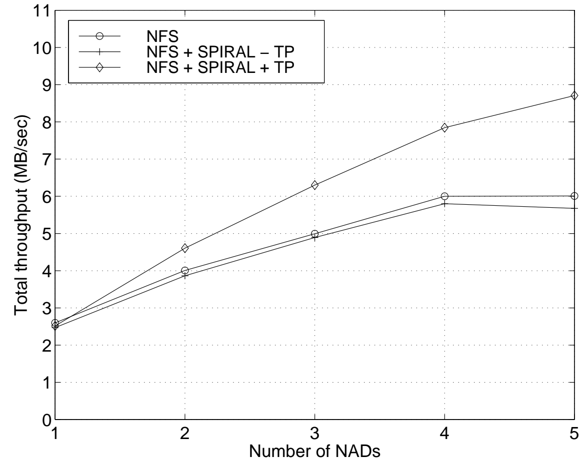

Figure 9. Throughput of NFS (TCP) for large file transfers

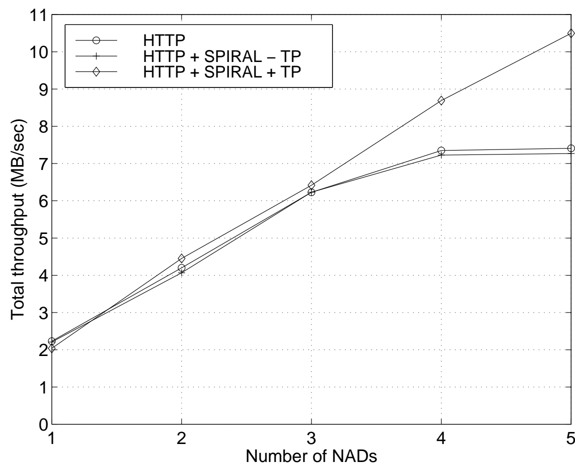

Figure 10. Throughput of HTTP for large file transfers

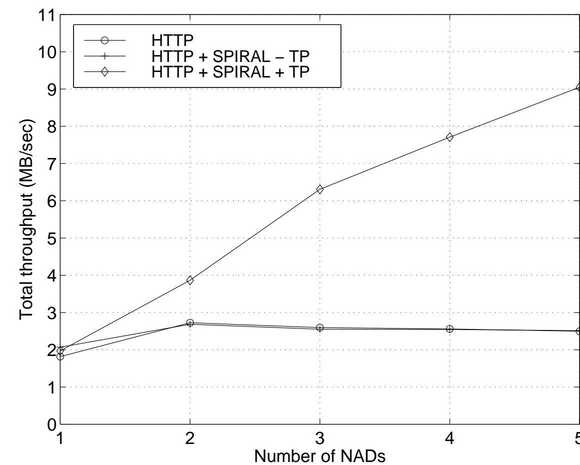

Figure 11. HTTP with limited interconnect Bandwidth

ure 11 shows that the throughput of the normal apache server is limited at around 2.5MB/s since all the data have to go through the server. The throughput of SPIRAL remains about the same since the aggregate network bandwidth of the system (server and the NADs) grows linearly with the number of NADs when the NADs have third-party transfer capability.

3.1.3. CPU Usage. In this experiment, we compare the CPU usage on the server with and without SPIRAL by measuring the remaining CPU power using the dhrystone benchmark [25]. Running the benchmark on the server during idle time gives 628,930.8 dhrystones per second. We ran the benchmark on the server while the clients are retrieving large files from all the five disk machines at the same time. Table 1 lists the results reported by dhrystone. For fair comparison, the SPIRAL throughput is limited to about the same as that of the normal server, that is, around 7MB/s for HTTP and 6MB/s for NFS. The results show that with normal NFS/HTTP server, the server CPU is heavily loaded and there is little processing power (smaller number of dhrystones) left for other processing. With SPIRAL, there is much more processing power available at the same throughput, 4 to 5 times more than that with a normal NFS/HTTP server. Note that same amount of data are sent by the application and go through the server's network protocol stack in both cases. The savings of the CPU power mainly come from the reduced network processing and data copying (during data transfers from NADs to the server).

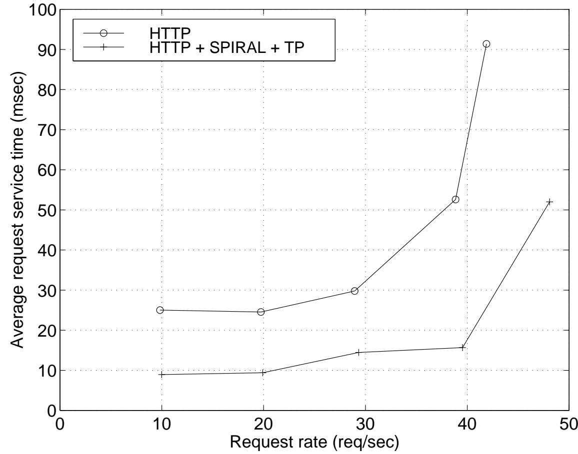

Figure 12. Average service time for small HTTP requests

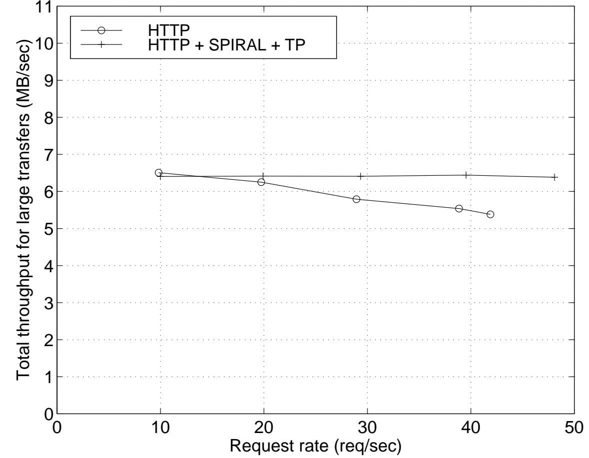

Figure 13. Total throughput for large HTTP transfers

Table 1. Remaining CPU processing power

	Dhrystones per second		
Application	Normal	SPIRAL - TP	SPIRAL + TP
NFS UDP	70,109.8	59,784.8	375,657.4
NFS TCP	46,285.6	42,612.1	226,295.5
HTTP	62,985.5	50,959.7	263,504.6

3.1.4. Memory Usage. In SPIRAL, large data transfers no longer result in the server's buffer cache being flushed. To show that SPIRAL can also increase the server's responsiveness for requests that do not use the third-party transfers directly (e.g., NFS GETATTR requests or HTTP requests for small files), we let one client repeatedly retrieve 250 64KB files whose TP attributes are not set from the server through HTTP while other clients are retrieving large files from all the five disk machines simultaneously. The small files are retrieved several times before the test to warm up the buffer cache on the server. The average service time for retrieving one 64KB file through HTTP from server to client (including network transfer time), is about 8.8ms in our system when there is no large file transfers. During the test, the total throughput for large file transfers in the background is limited to around 6MB/s to leave enough network bandwidth for small file transfers. Figure 12 shows the average service time for small file request at different small request rates. Figure 13 shows the corresponding total throughput for the large file transfers. When the rate for small file requests is low, the average service time with SPIRAL is close to the ideal result while the average service time without SPIRAL is 2.5 times longer. As the request rate increases, the results are not only affected by the memory usage on the server, but also impacted by the CPU load on the server. The SPIRAL HTTP server manages to provide better service times while serving small requests at a higher rate. The results in figure 13 show that the total throughput for large file transfers in a standard server decreases at higher small request rates while the SPIRAL server could continue serving large files at 6MB/s.

3.2. Requests of different sizes

The results from previous measurements demonstrated the benefits of SPIRAL for transferring large files. In this experiment, we study the sensitivity of these benefits to different request sizes. Figure 14 and figure 15 give the throughput of the system when the clients issue requests for files of different sizes from all five disk machines through NFS(UDP) and HTTP respectively.

To prevent disk accesses from being the bottleneck for tests with small file sizes, we use a RAM disk on each disk machine to ensure that the server's CPU is saturated in these measurements. Because of this, the maximum total throughput we could achieve here are higher than those in figure 8 and figure 10. In figure 14, the maximum throughput for SPIRAL is able to exceed 100Mbps because the Ethernet switch allows each disk machine to send data to clients at 100Mbps simultaneously. Results in figure 14 show that for NFS over UDP, SPIRAL is able to achieve a throughput of about 70% higher than normal

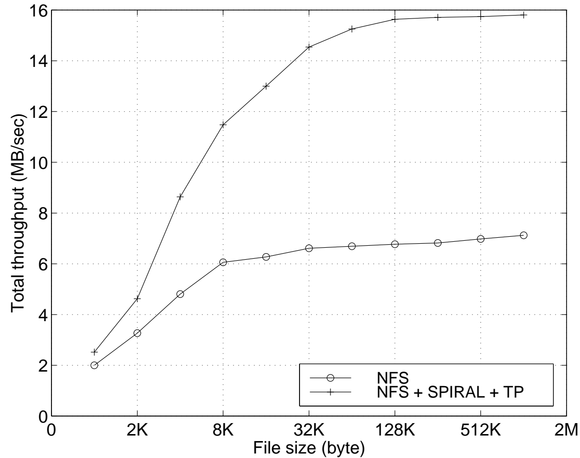

Figure 14. Throughput of NFS (UDP) with different file sizes

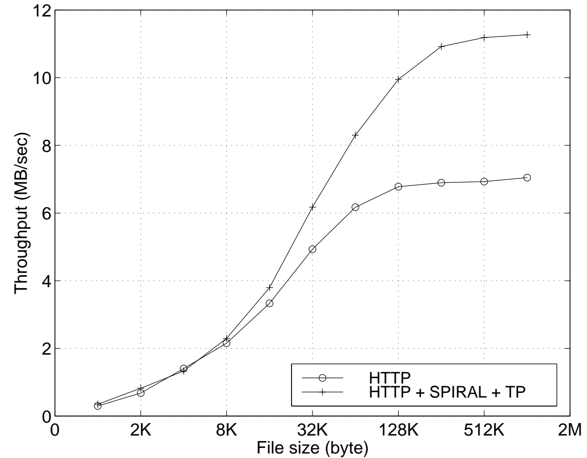

Figure 15. Throughput of HTTP with different file sizes

Table 2. Number of NFS operations

File size in experiment	NFS Lookup	NFS Read
4KB	12213	11651
128KB	598	16848

NFS server for files of 4KB size. We also notice that SPIRAL shows significant performance improvement at even small file sizes. Similarly, for HTTP, we observe that SPIRAL shows considerable performance improvements for file sizes larger than 32KB.

From these results, we can see that SPIRAL would be most useful for workloads where data transfer dominates, such as those of networked video servers. Video request sizes are typically large and the initialization and setup cost for each video transfer is relatively small. For example, the workload for the video server in [2] shows the average number of bytes transfered per play is 29.8M. More generic NFS or Web server workloads, however, will benefit less from SPIRAL since data movement contribute to a smaller fraction of the total workload.

For the NFS trace analyzed in [8], read requests account for around 20% of the total number of requests and around 30% of the CPU cycles on the server. There, the costs of NFS operations were measured by recording code-path execution times when the requested data are cached on the server. This means that cost of each NFS operation reported in [8] does not include the cost of disk IO operations. When the requested data need to be fetched over the network from the NADs, we expect to see a higher relative cost for read operations due to network processing. To confirm this, we measured the number of NFS operations for normal NFS server in the experiment corresponding to figure 14 with file sizes of 4KB and 128KB. Table 2 lists the number of NFS lookup and read operations carried out in a fixed amount of time. The number of other NFS operations during the experiment is very small and is negligible. If we assume the cost of a lookup is l, an NFS read is r, since the server's CPU is fully loaded in both the cases, then from the table, we have $12213 * l + 11651 * r \approx 598 * l + 16848 * r$, which gives $r = 2.25 * l$. Thus, we observed the cost of an NFS read operation (each NFS read request reads 4KB data in the experiments) to be roughly about 2.25 times as expensive as a lookup operation when data need to be read from NADs. A comparison of data in [8] shows that, without IO accesses, a 4KB NFS read is around 1.5 times as expensive as an NFS lookup. This demonstrates that even if read operations remained at 20% in NFS workloads, the relative CPU cost of these operations is likely to be higher than 30% when NADs are employed. According to results shown in figure 14, SPIRAL can reduce the cost of handling read requests to less than half on the server. Based on these data, we conclude that SPIRAL will provide 15% or higher performance improvement for generic NFS workloads.

3.3. Realistic workload

To see how the system performs in practice, we ran the Ziff Davis Media WebBench 4.1 benchmark program on our system. We ran one WebBench client on each of the three client machines under Win98. Each WebBench client used 15 threads and each test ran for 5 minutes. The standard workload tree that comes with WebBench contains 6,160 files and requires approximately 61MB of space. We distributed these files evenly across all five disk machines. The workload tree uses 10 file sizes ranging from 223 bytes to 529KB. The majority of these files are HTML and GIF files. The standard static workload file that comes with

Table 3. Original file size distribution among requests

File size (byte)	223	735	1522	2895	6040	11426	22132	41518	87360	541761
Distribution (%)	20	8	12	20	14	16	7	0.8	0.1	0.1

Table 4. WebBench results (N=NORMAL S=SPIRAL)

	Server memory(MB)	With HTTP 1.1	With video	Request per sec	Throughput (MB/sec)	Data fetched from disk to server(MB)
N	64	No	No	251	1.51	250
S	64	No	No	252	1.51	17
N	128	No	No	253	1.51	65
S	128	No	No	255	1.55	17
N	64	Yes	No	789	4.75	650
S	64	Yes	No	844	5.09	17
N	128	Yes	No	1020	6.01	65
S	128	Yes	No	849	5.12	17
N	64	Yes	Yes	248	5.66	1550
S	64	Yes	Yes	396	6.93	22
N	128	Yes	Yes	348	6.62	1100
S	128	Yes	Yes	399	7.76	18

Table 5. File size distribution among requests with video files

File size (byte)	223	735	1522	2895	6040	11426	22132	41518	87360	541761	32M
Distribution (%)	20	8	12	20	14	16	7	0.8	0.1	0.05	0.05

WebBench was used to generate the HTTP requests (the static workload contains only requests for static files). Table 3 gives the distribution of file sizes among the generated requests. To show the performance of the system both when all data can and cannot be cached in the server's memory, we ran each test with two memory configurations on the server: 64MB and 128MB. For each configuration, we ran the benchmark when all HTTP requests generated are HTTP 1.0 requests, and when 50% of all requests are HTTP 1.1 requests using persistent HTTP connection. The minimum number of requests per connection for HTTP 1.1 requests is set to 1 and the maximum is set to 10. Requests can be pipelined in each persistent connection and the minimum and maximum number of pipelined requests are set to 1 and 10 respectively. We set the file size threshold for third-party transfer to 8KB for SPIRAL measurements, which means that only those files exceeding 8KB will be sent by the NADs to clients directly while smaller files are served in the normal way. Each measurement started with a cold cache on the server. Table 4 lists the results of our measurement including two overall server scores reported by WebBench (request per second and throughput in MB/sec) as well as the total amount of data fetched from the disks to the server. We can see that when the workload contains only HTTP 1.0 requests, the results of SPIRAL and normal system are very close since each HTTP 1.0 request uses a separate TCP connection and the server's CPU is saturated for handling the connections. With workload containing HTTP 1.1 requests (With HTTP 1.1), the normal system performed better when the server is equipped with 128M memory, in which case the whole workload tree can be cached in the server's memory (the total amount of data fetched from the disks to the server roughly equals the size of the workload tree). This is expected since there is little advantage for redirecting the packets to the NADs when the server can serve all the requests from its memory directly. When the server has only 64M memory, SPIRAL outperforms the normal system because the latter has to constantly fetch data from the disks due to limited cache size. For SPIRAL, only files smaller than 8KB need to be fetched to the server and they are not purged from the cache due to requests for larger files.

In the table we also show what happened when a small number of large video file transfers are introduced into the workload. Table 5 gives the distribution of file sizes among the requests after we modified the workload tree by adding some 32MB MPEG video files. The results (With video) in table 4 indicate that even though there is only one video request out of around 2000 requests, these video requests can reduce the number of requests completed per second

substantially. Compared with the normal system, SPIRAL performs significantly better with 64MB memory on server and noticeably better with 128MB when the workload contains large file transfers.

4. Related Work

In NASD [8], each disk manages its data layout and exports an object interface. File operations such as read and write go directly to the disk, while other operations such as namespace and access control manipulation go to the server. The security is protected by granting time-limited tickets to the clients. Their results show significant improvement on the system's scalability. The clients are aware of the disks and client OS/applications need to be modified to access objects on the disks directly. It's likely that, given the same hardware configuration, approaches such as NASD will perform better than SPIRAL because the clients access the disks directly and disks in NASD support writes and attribute read requests. This is the price that SPIRAL pays in order to achieve client-side transparency and to minimize changes to the NAD OS. Some other approaches proposed to offload computation to smart storage devices include active disks and IDISK [1, 21, 14].

The derived virtual device (DVD) model [18] proposed in the Netstation project provides a mechanism for safe shared device access in an untrusted environment by creating DVDs and managing them through a network virtual device manager. The proposed third-party transfer scheme using DVDs is similar to that in NASD. The Linux NBD that is used in our prototype is similar to their virtual Internet SCSI adapter [17].

In LARD [20, 5], outgoing data are directly sent to the clients by the back-end server without going through the front-end server through a TCP handoff protocol. A TCP handoff protocol is used to achieve client side transparency. After a TCP connection is handed off to a back-end server, the front-end server still receives all the incoming data but forwards them at lower network stack layer to back-end servers. However, LARD assumes a cluster configuration where each back-end server is capable of serving requests independently. For NADs, such a scheme requires porting file system and applications (such as NFS/HTTP server) on the disks, which essentially turns each NAD into a small server.

In Slice [4], a request switching filter interposed along the network path between the client and the storage server routes file requests based on request type and parameters. Slice focuses on NFS over UDP and requires changes to the network routing components between the client and the server. Both the packet redirector in SPIRAL and the network switch filter (call μproxy) in Slice manipulate packets at datalink level based on application-level information but in the opposite direction.

There are many approaches to implement request distribution across a number of servers [6, 13, 7, 26]. Since packet redirection in SPIRAL happens at the datalink layer, SPIRAL can be used in conjunction with these solutions seamlessly. On the other hand, although we focused on third-party transfer with NADs in this paper, the packet redirection scheme used in SPIRAL can also be deployed among the servers to achieve similar goals as those in these solutions.

Currently SPIRAL does not support third-party transfer for incoming data from clients. The Zebra striped network file system [10] allows writes to the storage server to append to the log without authentication. The written blocks don't become part of the visible file system until the file manager updates the metadata. The storage server in Zebra only has to deal with file system requests. Supporting third-party transfer for incoming write requests in SPIRAL is more complicated since the NADs need to handle incoming data from different applications.

What SPIRAL does essentially is to separate data and control transfer at the datalink level from the server side. Separating data and control transfer to achieve better scalability and to minimize shared network resources usage was discussed in [23]. Such a separation was also advocated by the Mass Storage System Reference model [12]. Several techniques have been proposed to remove copies from the data path. IO-Lite [19] introduces a unified IO buffering and caching system to eliminate data copying and multiple buffering. The Direct Access File System (DAFS) [16] promises high-performance network-attached storage over direct-access transport networks by taking advantage of user-level network interface standards. What distinguishes SPIRAL from other approaches is that SPIRAL achieves client-transparency and reduces data transfers between the server and the NADs at the same time.

5. Conclusions & Future Work

In this paper, we presented a client-transparent third-party transfer scheme for storage systems with NADs. We showed that third-party transfer over reliable transport protocol such as TCP can be supported efficiently with datalink level packet interception. Our approach is based on networked storage devices with block-based interfaces (such as iSCSI) and does not require porting file system functionality onto the device. Results for NFS and HTTP on a Linux PC-based prototype system demonstrated the effectiveness of SPIRAL.

Acknowledgements

This research is funded in part by an NSF Career Award, NSF research grants CCR-9901640, CCF-0098263 and by the State of Texas.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks. In *Proceedings of ASPLOS-VIII*, October 1998.

[2] J. Almeida, J. Krueger, D. Eager, and M. Vernon. Analysis of educational media server workloads. In *Proc. of NOSS-DAV*, June 2001.

[3] W. Almesberger, A. Kuznetsov, and J. H. Salim. Differentiated services on linux. In *Proceedings of GlobeCom*, pages 831–836, December 1999.

[4] D. Anderson, J. Chase, and A. Vahdat. Interposed request routing for scalable network storage. In *Proc. of the 4th Symp. on OSDI*, Oct. 2000.

[5] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for P-HTTP in cluster-based web servers. In *Proceeding of the 1999 Annual Usenix Technical Conference*, June 1999.

[6] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. In *Proceedings of Usenix Symposium on Internet Technology and Systems*, February 1999.

[7] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29(8–13):1019–1027, 1997.

[8] G. A. Gibson and et al. File server scaling with network-attached secure disks. In *Proceedings of the ACM SIGMETRICS*, June 1997.

[9] R. Haagens and A. Romanow. TCP ULP Framing/iSCSI Framing. http://www.haifa.il.ibm.com/satran/ips/Randy-Haagens-Framing_r1.pdf.

[10] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 29–43, December 1993.

[11] J. S. Heidemann and G. J. Popek. File system development with stackable layers. In *Transactions on Computing Systems*, 1994.

[12] IEEE Storage System Standards Working Group (Project 1244). *Reference Model for Open Storage Systems Interconnection, Mass Storage Reference Model Version 5*, September 1994.

[13] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27(2):155–164, 1994.

[14] K. Keeton, D. A. Patterson, and J. M. Hellerstein. The case for intelligent disks (IDISKs). *SIGMOD Record*, 27(3):42–51, September 1998.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Henenessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 148–159, May 1990.

[16] K. Magoutis, S. Addetia, A. Fedorova, M. I. Seltzer, J. S. Chase, A. J. Gallatin, R. Kisley, R. G. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system. In *Proceedings of USENIX 2002 Annual Technical Conference*, pages 1–14, June 2002.

[17] R. V. Meter, G. Finn, and S. Hotz. VISA: Netstation's Virtual Internet SCSI Adapter. In *Proceedings of ASPLOS-VIII*, October 1998.

[18] R. V. Meter, S. Hotz, and G. Finn. Derived Virtual Devices: A secure distributed file system mechanism. In *Proc. of the 5th NASA Conf. on Mass Storage Systems and Technologies*, Sept. 1996.

[19] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *Rice University Tech. Report: TR97-294*, 1997.

[20] V. S. Pai and et al. Locality-aware request distribution in cluster-based network servers. In *Proc. of ASPLOS-VIII*, Oct. 1998.

[21] E. Riedel, G. Gibson, and C. Faloustos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th VLDB Conference*, August 1998.

[22] Sun Microsystems Inc. NFS:Network File System Protocol Specification. *IETF RFC 1094*, March 1989.

[23] C. Thekkath, H. Levy, and E. Lazowska. Separating data and control transfer in distributed operating systems. In *Proceedings of ASPLOS-VI*, October 1994.

[24] A. Vahdat, T. Anderson, and M. Dahlin. Active names: Programmable location and transport of wide-area resources. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[25] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27:1013–1030, 1984.

[26] C. Yoshikawa and et al. Using smart clients to build scalable services. In *Proceedings of the 1997 Annual USENIX Technical Conference*, January 1997.