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

Xiaonan Ma and A. L. Narasimha Reddy
Department of Electrical Engineering
Texas A & M University
College Station, TX 77843-3128
{xiaonan, reddy}@ee.tamu.edu

Abstract

In this paper, we present SPIRAL [1], a simple scheme for supporting third-party transfer on storage systems with network-attached disks (NADs). SPIRAL is totally transparent to clients and retains the linear block interface of current disks. In SPIRAL, there is no need to port file-system and application-level functionality to disks and few changes are required to the server's OS and applications. To illustrate our approach, we implemented a prototype system on PCs running Linux. We present 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 [19, 15] 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 in improving the device and system interconnection speeds. Further, IP based storage systems are expected to consolidate the network wiring infrastructure of data centers. A number of early research projects [6, 17, 13] have pointed out the benefits of directly attaching storage devices on the network. Improved server scalability and faster data transfers are some of the benefits of these devices.

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 transfers 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 data of 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.

Several researchers have pointed out the possibility of improving the performance of the servers for such workloads through third-party transfers from network attached devices (NADs) [6, 17, 10]. Third-party transfer is a data transfer mechanism where the party initiating the transfer is neither the source nor the sink for the data. Early work at CMU [6] showed that third-party data transfers can greatly improve the scalability of file systems. However, existing approaches normally require the clients to be modified to communicate with disks directly, or porting of file-system and application-level functionality onto the disk. It is also shown that rigid deployment of third-party transfers may not be universally beneficial. In some workloads, network-attached disks can reduce data cache hit rates sufficiently to offset the benefits of improved parallelism in network transfers [13].

In this paper, we propose SPIRAL, a new scheme for third-party transfers utilizing NADs. SPIRAL enables NADs to send data to clients directly over the network instead of through the server. It has the following characteristics:

- It is **client-transparent**. There is no need to modify the OS and applications on the client side.

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

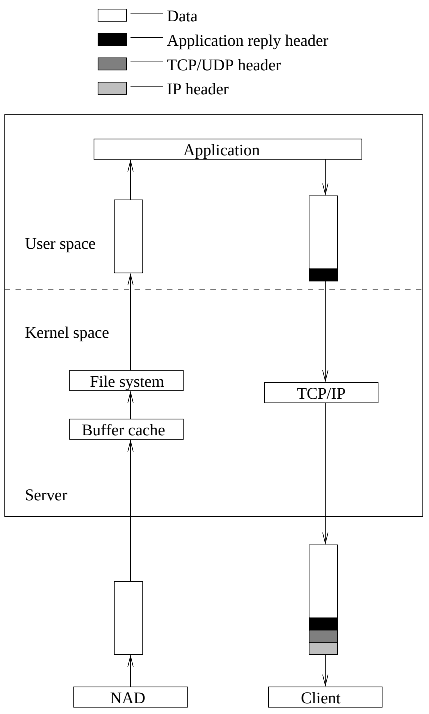

Figure 1: Data flow for read in traditional system

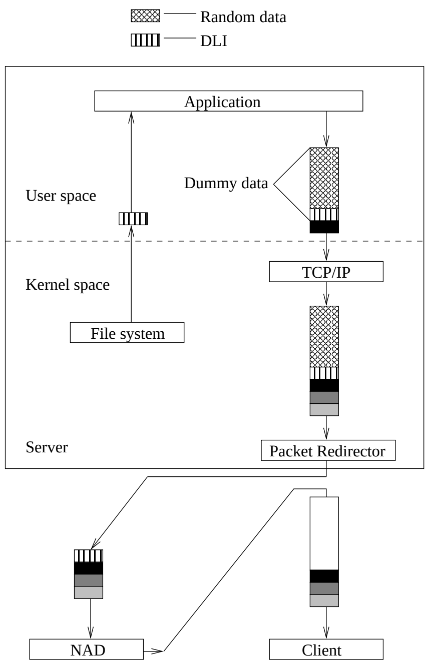

Figure 2: Data flow for read in SPIRAL

- It uses existing **block device interface**, which allows the simplest reuse of existing file system and operating system technology.
- It keeps the software layer on the disk as thin as possible for efficient utilization of disk resources. No file system or application-level support is required on disks.
- It requires little modification to applications on the server and supports both UDP and TCP.

The rest of the paper is organized as follows. Section 2 presents the design rationale 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. In Section 4, we compare various aspects of SPIRAL with related work. Section 5 concludes the paper.

2 Design

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. By caching metadata and small files centrally at the server, we can achieve higher buffer cache hit rates than possible with similar size caches distributed among multiple NADs [13]. 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 employ 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 [23], Multiview Storage Systems [14], and other namespace based approaches.

We assume that NADs support a linear block level interface as 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 is 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, `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 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 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 then 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 shrinked packets are then redirected to the disks. Note that unlike in [2] where packets are redirected by manipulating packet headers, shrinked packets in SPIRAL are actually tunneled to the disks with the packet headers treated as normal payload. Tunneling the packets simplifies the receiving code on the disks and allows us to leverage the existing security protection scheme of the communication channels between the server and the disks. We will use the term "redirect" and "tunnel" interchangeably in this paper. After a disk 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 disk 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. Since 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 to clients. Clients send data requests to the server, the server fetches the data from the disk 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

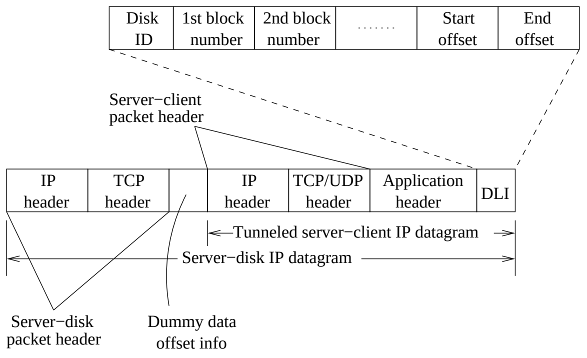

Figure 3: Redirected packet structure

dummy data are acting as place holders and does 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) the data copies from the device to the server memory are avoided; (2) the server's network stack buffers are released sooner since dummy data is no longer needed after 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 shrinked packet is tunneled and also shows the structure of a typical DLI.

The advantage of intercepting packets at datalink layer is that all the redirected packets have complete IP and TCP/UDP header information, so there is no need to maintain transport-level state information for connections with clients on the disk side. All network issues such as acknowledgements and retransmissions are handled by the server. Since these packets also contain application-level reply headers, the disks 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 disk data transfers while performing all application-level processing without incurring the overhead of frequent connection handoffs. As a result, the disks only need to perform operations which are application-independent. This simplifies the design and implementation of our system and minimizes the overhead of setting up each third-party transfer.

In SPIRAL, all dummy data still need to be copied from user space kernel network buffers even though most of the dummy data will not be used. The load on the server can be further reduced if this data copy can be eliminated. One possible approach is to let applications on server send out redirection information directly to the disks instead of intercepting the reply packets at datalink layer. However, this requires more modifications to the server application and will only work with connection-less protocols, such as UDP. For TCP, it requires either a TCP handoff protocol similar to that proposed in LARD [18], 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.

Compared with a traditional system, SPIRAL offers several main benefits. First, it greatly reduces the amount of data traffic between the server and the disks. The server only needs to read the metadata for building the DLIs. The bandwidth required for transferring the metadata is 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 disks is the bottleneck. Second, since the data no longer go through the server's buffer cache, the memory usage on the server is reduced. This will decrease the number of page faults on the server and increase the cache hit rates for other requests. Third, even though our approach does not reduce the server's workload on building all reply packets and maintaining the state information for each connection, the actual work of sending all the packets out to the clients is now distributed among all the disks, which reduces the CPU load and the number of interrupts on the server.

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 disk? 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 disks? 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. At this level, outgoing packets that belong to applications supporting third-party transfer, but do not contain dummy data are filtered out.

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 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.

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. One 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 using the reply's RPC XID and client IP address (the XID is guaranteed to be unique for each NFS request from one client). Another 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 byte 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 maintain 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 [8].

2.2 TCP related issues

Redirecting packets at datalink level allows SPIRAL to support applications where multiple requests are sent over a single connection (e.g., persistent HTTP and NFS over TCP) with much smaller overhead, compared with other client-transparent approaches such as TCP handoff [18] and TCP splicing [4]. It simplifies the disk-side processing, allows disks to support implementation-specific TCP options between server and clients directly in most cases even if they are not implemented on the disks. However, these advantages do not come without cost. There are several complications for supporting applications over TCP in SPIRAL.

2.2.1 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 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, we can safely assume that outgoing packets are delivered in order at datalink level. 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.

2.2.2 Incomplete DLI

Since TCP may break packets at any place in the data stream, it is possible for a packet to only contain partial DLI. In this case, we don't have enough information on how to shrink the packet and where to redirect it. This packet cannot be sent to the client because the incomplete DLI is part of the dummy data. This can be solved by holding the packet temporarily in the packet redirector until the next outgoing packet in the stream arrives and completes the DLI. Because the size of a DLI is much smaller than the average packet length, this only happens to a small fraction of all packets containing dummy data and has little impact on the performance.

However, holding a TCP packet may cause a potential deadlock if the server can not send out the next packet until new acknowledgement is received, and the client is waiting for the packet currently being held before sending out acknowledgement. Most likely, this would happen when the size of the TCP congestion window on the server side is only one Sender Maximum Segment Size (SMSS), for example, when the connection is just initialized. The probability for the deadlock to happen is very small since it requires both incomplete DLI and a small congestion window. To deal with this, if a packet has been held for longer than a certain timeout period, spoofed acknowledgements (ACKs) can be generated by the packet redirector masquerading as the client based on the most recent ACK received from the client. These ACKs will appear to the sender to be duplicated ACKs, and will trigger the sender's TCP fast retransmit algorithm. As a result, the congestion window is increased and the next outgoing packet in the stream will be sent. This does not violate the TCP semantics or reliability [20]. Another solution is to let the application pass DLIs to the packet redirector through an ”out-of-band” channel instead of embedding it in the dummy data.

2.2.3 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.

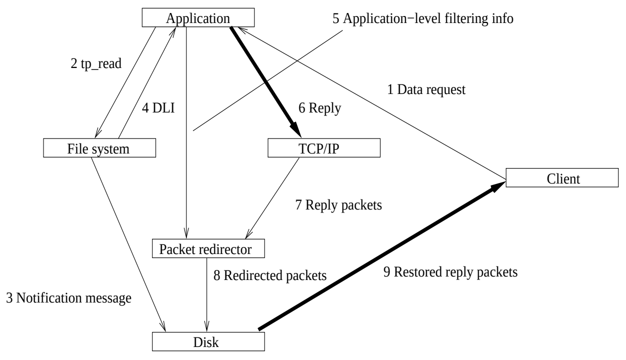

Figure 4: Third-party transfer procedure in SPIRAL

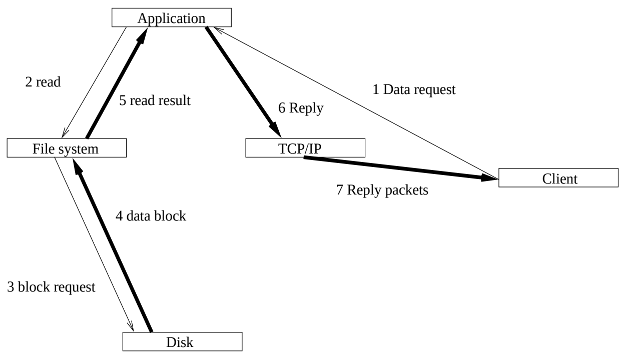

Figure 5: Data transfer procedure in a traditional system

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 disk 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 write file data in user buffer but only resolves DLIs, problems arise due to the delay between when `tp_read` resolves the DLI and when the disk 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 disk. All these could happen before the redirected packets arrive at the disk. In this case, the disk will read the wrong data and send them to the client based on the obsoleted information in the DLI.

SPIRAL solves this problem by letting `tp_read` send a short notification message to the disk 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 disk unnoticed before the notification message. When the disk receives the notification message, it marks these blocks as "pending", which means that the disk 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 disks 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 disk 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 disk 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 [12]. In practice, the release messages can be sent out to the disks 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 disk can start reading the data blocks into memory without having to wait for the redirected packets.

2.4 Multi-disk redirection

Sometimes a single packet may contain multiple pieces of dummy data whose corresponding file data reside on different disks. This could happen, for instance, when several files from different disks are required through a persistent HTTP connection, or when data in a single file are striped over several disks. There are several approaches to deal with such packets. The simplest may be just to let all the disks 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 disk. Alternatively, the host could split the packet into several IP fragments and send one to each of the disks. In such a case, the packets are processed on each disk independently and no data need to be transfered between the disks. 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 disk controller), then using `bmap` to resolve DLI is sufficient since the volume to device mapping will be done on the disks. 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 disks to perform access control tasks (such as file permission checking and client authentication). Also, because the disks 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 disks securely. This is not a limitation of SPIRAL because such secure communication channels between the server and the disks are required for any storage system with NADs, whether SPIRAL is used or not. In SPIRAL, the redirected packets are tunneled to disks as normal data payload, therefore no extra protection scheme is required.

2.6 Limitations

One limitation of SPIRAL is that it only works 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 disk directly also requires that disks 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. Another example is when an HTTP server serves dynamic pages or chooses to compress the data before sending them to the client.

Another limitation of SPIRAL is that it will not work if the packets are protected by encryption schemes such as IPsec or SSL. It may be possible to let the server share keys with the disks so that the disks could encrypt the data by themselves. Similar problem will be encountered by other third-party transfer schemes as well.

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 several 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.

2.8 Future application

Although we focus on third-party transfer between the NADs and clients in this paper, there is nothing that prevents the server from redirecting shrinked 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 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. For applications

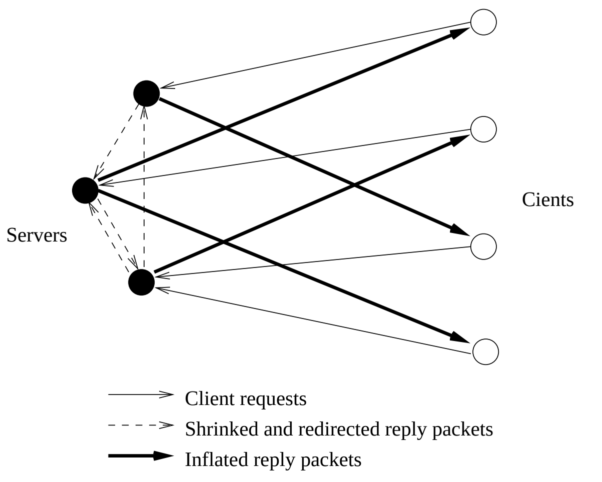

Figure 6: Third-party transfers with SPIRAL in cluster

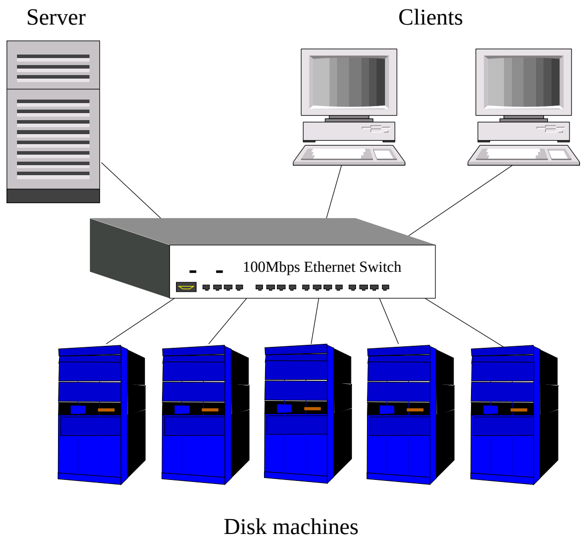

Figure 7: Testbed structure

which send multiple data requests over a single TCP connection (as in persistent HTTP), such a scheme can greatly reduce the data traffic among the servers without incurring the overhead of frequent connection handoffs.

3 Implementation

The structure of our testbed is shown in figure 7. The 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 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.

In the prototype implementation, we introduced a third-party transfer file attribute (called TP attribute) through a simple stackable file system layer [9]. 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. 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 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

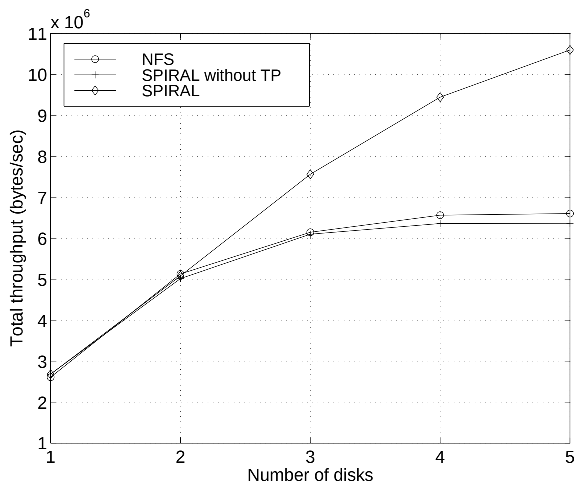

Figure 8: NFS UDP

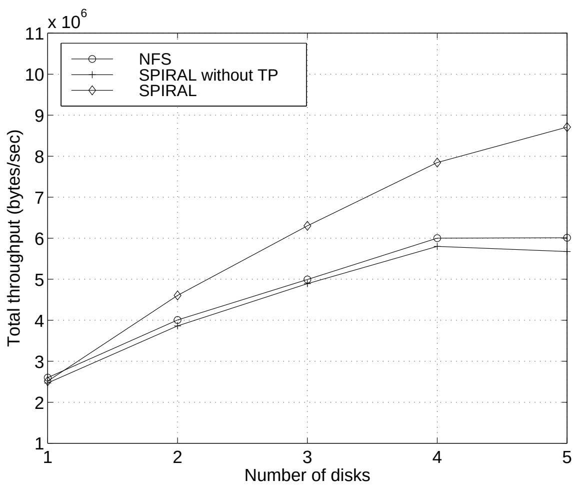

Figure 9: NFS TCP

on the server sequentially for each disk in the system. Figures 8 and 9 show the total throughput of the NFS server over UDP and TCP respectively with different number of disks. 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 (SPIRAL) scales almost linearly as the number of disks increases. With five disks the total throughput approaches the network bandwidth limit (100Mb/s) while the throughput for normal NFS is saturated at around 6MB/s.

We also show in the figure the results of SPIRAL when the TP attributes for the files are not set (SPIRAL without 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 by the RPC parser) will not be performed.

3.2 HTTP results

We did similar experiments for HTTP. The results are shown in figure 10. Again, the results show that SPIRAL allows improved scalability with the number of disks and the overhead on regular traffic (SRIRAL without TP) is negligible. We also note that with similar configurations, the HTTP results are a little better than the corresponding NFS 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 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. HTTP does authentication once per file compared to authentication per 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 disks is limited to 2.5MB/s using the Linux network QoS support [1]. Figure 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 disks) grows linearly with the number of disks when the disks have third-party transfer capability.

3.3 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 [24]. 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 disks 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 pro-

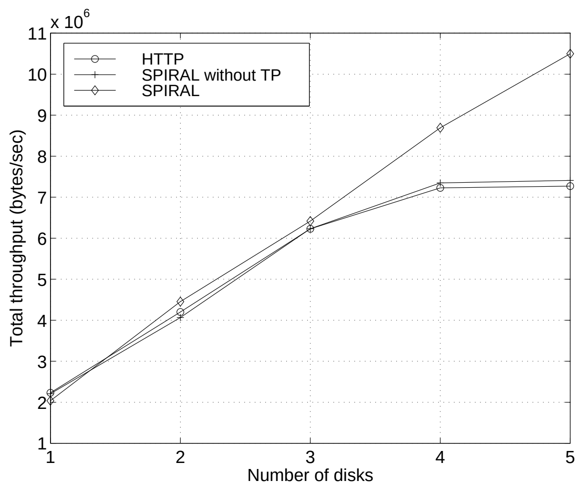

Figure 10: HTTP

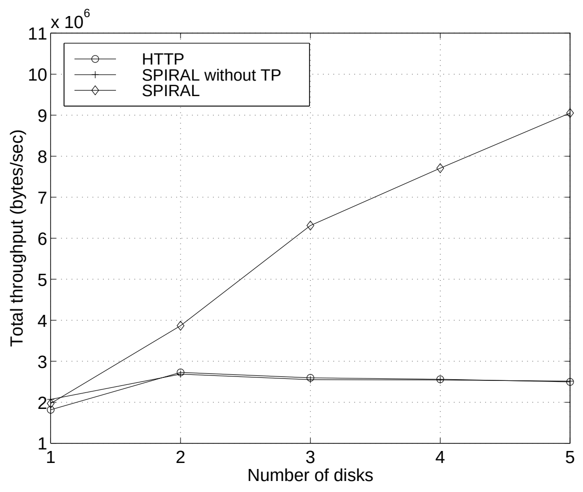

Figure 11: HTTP with limited interconnect Bandwidth

Table 1: Remaining CPU processing power

	Dhrystones per second		
Type	Normal	SPIRAL without TP	SPIRAL
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

cessing and data copying (during data transfers from disks to the server).

3.4 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 disks 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 small file, when there is no large file transfers, is about 8.8ms. 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.

In today's large multiprocessor servers, memory usage often becomes the system's bottleneck [21]. Solutions such as in-memory compression have been applied to reduce this problem. By reducing the amount of memory usage on the server, SPIRAL helps in improving the scalability of the system. In order to show the advantage of SPIRAL on the system's memory usage in a more direct way, we developed a simple benchmark program that sends out SMSS size pre-built UDP packets through a raw socket with the IP_HDRINCL option set. Using the IP_HDRINCL option helps to minimize the processing overhead. The benchmark uses a kernel thread to send out all the packets to avoid the overhead of context switches between user space and kernel space. The goal of the benchmark is to perform as much memory access as possible with minimum CPU load. Sending each packet in the benchmark requires one memory copy in the kernel and one DMA transfer be-

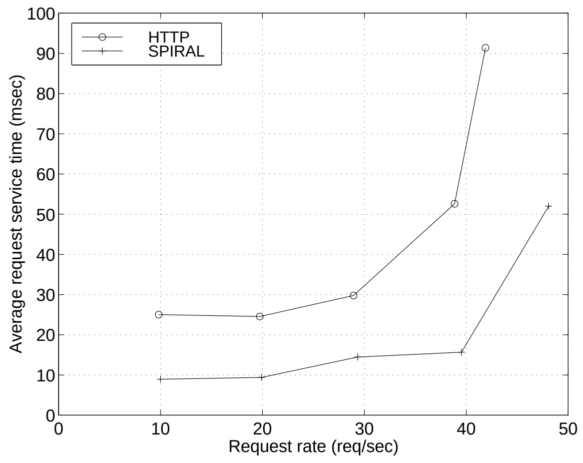

Figure 12: Average service time for small HTTP requests

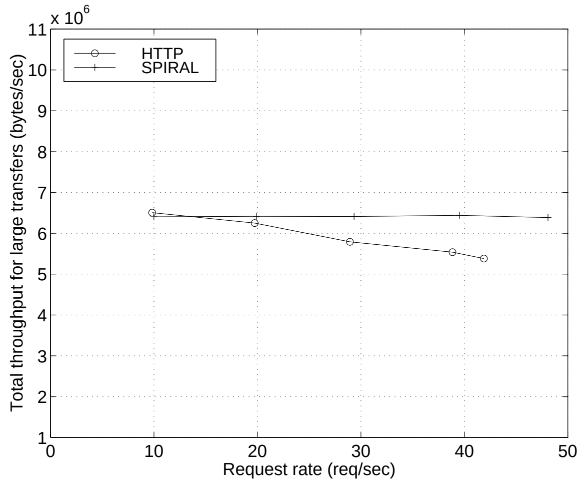

Figure 13: Total throughput for large HTTP transfers

Table 2: Results for data transfer experiment

Type	Large file transfer throughput (MB/s)	Raw packet throughput (MB/s)
NFS-UDP	5.71	4.61
SPIRAL without TP	5.74	4.40
SPIRAL	8.09	11.9

tween the memory and the network interface card (NIC). We installed a second NIC on the server and connected one client machine to the server directly, i.e., not through the Ethernet switch. In the experiment, we let the benchmark send out packets through the second NIC with large file transfers using NFS/UDP through the first NIC at the same time. This was done to prevent the benchmark traffic and the NFS transfers from competing for network bandwidth. The result in table 2 shows that without SPIRAL, the benchmark can only send out packets at about 4.6MB/s. With SPIRAL, we can keep the NIC saturated at its bandwidth limit (100Mb/s) while simultaneously achieving a higher throughput for large file transfers. This demonstrates that with SPIRAL the system can sustain higher memory transfers.

4 Related Work

In NASD [6], 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. However, this approach is not client-transparent. The clients are aware of the disks and client OS/applications need to be modified to access the disk object directly.

The derived virtual device (DVD) model [17] 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 [16].

One approach for distributing requests among servers while maintaining transparency to clients is proposed in LARD [18]. In LARD, the front-end servers use a hash function to distribute requests to other back-end servers. 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. Outgoing data are directly sent to the clients by the back-end server without going through the front-end server. 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 turning each disk into a small server.

An extension of LARD which supports persistent HTTP connection is presented in [3]. A back-end forwarding mechanism which forwards replies among the back-end nodes is proposed to avoid the complexity and overhead of multiple TCP handoffs.

In Slice [2], 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 [4, 11, 5, 25, 7]. 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.

5 Conclusions & Future Work

In this paper, we presented a server-side third-party transfer scheme for storage system with NADs. The scheme is client-transparent and requires little change to the applications running on the server. 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 the block device interface 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.

In the future, we plan to extend the datalink-layer packet redirection scheme in SPIRAL to support third-party transfer among a cluster of servers.

References

[1] W. Almesberger, A. Kuznetsov, and J. H. Salim. Differentiated services on linux. In *Proceedings of Globecom'99*, pages 831–836, December 1999.

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

[3] 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.

[4] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. *Proc. of Usenix Symp. on Internet Technology and Systems*, Feb. 1999.

[5] 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.

[6] G. A. Gibson and et al. File server scaling with network-attached secure disks. *Proc. of the ACM Sigmetrics*, Jun. 1997.

[7] S. Gupta and A. L. N. Reddy. A client oriented, IP level redirection mechanism. In *IEEE Infocom Conf.*, March 1999.

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

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

[10] R. W. Hosrt. TNet: A reliable system area network. *IEEE Micro*, pages 1–9, Feb. 1995.

[11] 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.

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

[13] G. Ma and A. L. N. Reddy. An evaluation of storage systems based on network-attached disks. *Proc. of ICPP Conf.*, Aug. 1998.

[14] X. Ma and A. L. N. Reddy. MVSS: Multiview Storage System. *Proc. of ICDCS*, Apr. 2001.

[15] R. V. Meter. A brief survey of current work on network attached peripherals(extended abstract). *Operating Systems review 30,1*, Jan. 1996.

[16] R. V. Meter, G. Finn, and S. Hotz. VISA: Netstation's Virtual Internet SCSI Adapter. *Proc. of 8th ASPLOS*, Oct. 1998.

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

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

[19] J. Satran and et al. iSCSI draft standard. www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-03.txt.

[20] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. Tcp congestion contorl with a misbehaving receiver. *Computer Communication Review*, 29(5), 1999.

[21] Serverworks, Inc. http://www.serverworks.com/.

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

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

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

[25] C. Yoshikawa and et al. Using smart clients to build scalable services. *Univ. of California, Berkeley Tech. report*, 1996.