A Fast Connection-Time Redirection Mechanism for Internet Application Scalability*

Michael Haungs[1], Raju Pandey[1], Earl Barr[1], and J. Fritz Barnes[2]

[1] Center for Software Systems Research
Department of Computer Sciences
University of California, Davis
{haungs,pandey,barr}@cs.ucdavis.edu
[2] Vanderbilt University
j.fritz.barnes@vanderbilt.edu

Abstract. Applications that are distributed, fault tolerant, or perform dynamic load balancing rely on redirection techniques, such as network address translation (NAT), DNS request routing, or middleware to handle Internet scale loads. In this paper, we describe a new connection redirection mechanism that allows applications to change end-points of communication channels. The mechanism supports redirections across LANs and WANs and is application-independent. Further, it does not introduce any central bottlenecks. We have implemented the redirection mechanism using a novel end-point control session layer. The performance results show that the overhead of the mechanism is minimal. Further, Internet applications built using this mechanism scale better than those built using HTTP redirection.

1 Introduction

Providing Internet services is increasingly difficult. Popular web sites must effectively handle current user request loads and prepare for increased future loads as their site gains popularity and the Internet audience, in general, grows. Excessive down times or user delay are unacceptable as Internet users will quickly turn elsewhere for their content.

To handle typical web loads, Internet server solutions must provide substantial computational power, I/O throughput and network bandwidth. This is accomplished either by using a high-end server or by clustering commodity workstations. Clustering is by far the most popular solution due to its cost effectiveness and potential for scalability. One of the main challenges with server clusters is distributing load while exporting a single name. Distributing load increases scalability. A single, easily remembered name for the service is necessary for end-user usability. Connection redirection, which maps the exported service name to a single server in the cluster, is the primary way to distribute load while still presenting a single name to the user.

* This research is supported in part by NSF grants CCR-00-82677 and CCR-99-88349.

Ideally, a redirection mechanism should be able to redirect requests to servers that are distributed in a wide area network and thereby allow servers to cope with network congestion and to exploit geographical, temporal and other locality properties for load balancing. Since popular server clusters must scale to millions of simultaneous requests, redirection mechanisms should not introduce any bottlenecks or failure points. A server cluster that provides short, predictable client latencies increases user-perceived quality [4]. Redirection mechanisms, thus, must allow servers to smoothly adapt to dynamic conditions such as flash crowds or machine failures. Finally, redirection mechanisms must be compatible with existing Internet protocols, incrementally deployable as sweeping changes to the Internet's infrastructure are unrealistic, and transparent so that existing applications, such as web and ftp servers, can take immediate advantage of them.

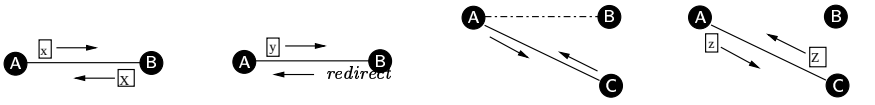
Existing applications use many techniques [3, 6, 8, 13] for dynamically redirecting connections: among these are application-layer protocols, name resolution manipulation, and network packet rewriting. Each solution varies in its client transparency, responsiveness to dynamic conditions, performance, and scalability. DNS redirection is transparent, but only provides coarse load balancing [6] and incurs increased lookup delay [15]. HTTP redirection is scalable, but not application independent. Network packet rewriting is fast and transparent, but limits clustering to servers within a local area network. Also, packet rewriters may become a central point of failure.

In this paper, we present a novel connection-time redirection mechanism, called *Redirectable sockets* (RedSocks). RedSocks adds a redirection operation to the communication channel API that allows a server to redirect channels as needed. During periods of contention, a server sheds load by invoking this operation. RedSocks can redirect communication channel end-points among both LAN and WAN separated servers. This provides servers with a flexible way to respond to network congestion. It is a protocol-based solution that aims at providing a general, application independent redirection facility. Unlike network packet rewriting, RedSocks only requires a single packet manipulation to redirect communication, so it does not introduce a chokepoint. RedSocks is backwardly compatible with client applications making it incrementally deployable.

In Section 2, we describe the `redirect` operation and give examples of its utility. Next, we present our implementation, which is highlighted by our novel session-layer protocol, called *end-point operation protocol* (EOP), that provides communication channel end-point control. In Section 4, we compare RedSocks directly to HTTP redirection and show a significant reduction in server redirection latency. In Section 5, we discuss related redirection mechanisms. Finally, we discuss future research directions.

2 Redirectable Sockets

Redirectable sockets provides a communication abstraction suitable for use in a variety of server architectures. In this section, we discuss the semantics and usage of RedSocks.



(a) Communication channel (b) Redirect system call (c) New connection creation (d) Communication on new channel

Fig. 1. Redirect end-point B to C

2.1 Definition

Our approach is to provide a primitive, `redirect`, that allows applications to redirect communication channels at connection time. We describe the semantics of a `redirect` system call through a simple example.

As shown in Figure 1(a), a communication channel exists between two nodes, A and B. The application at B then executes `redirect` in order to change the "B" end-point of the channel to C (see Figure 1(b)). C may define an end-point at A, B, or at a different host. The `redirect` system call creates a new channel between A and C, and removes the channel between A and B as shown in Figures 1(c)–(d).

There are two ways to handle the request made by A in Figure 1(b). B has the option to send a response to the request y during the `redirect` system call (See Below). However, if B does not, then A must resend its previous request.

```
s=socket( SOCK_EOP_STREAM );
accept();
if (earlyRedirect())
    redirect(s, NULL, endPointID);

recv(s, request);
response = process(request);

if (lateRedirect())
    redirect(s, response, endPointID);
else
    send(s, response);
```

(a) Server

```
s = socket( SOCK_EOP_STREAM );

REDO:
send(s, request);

if (recv(s, response) < 0 )
{
    if (errno == EREDO)
        goto REDO;
    reportError(errno);
    exit(1);
}
```

(b) Client

Fig. 2. RedSocks Sample Psuedo-Code.

2.2 Usage

An application can use the `redirect` system call to perform the redirect operation. This system call has the following general form: *redirect(data, EndPointID)*. The `redirect` operation first ensures that the data, if given, is delivered. Next,

the socket end-point belonging to the caller of the `redirect` system call is redirected to `EndPointID`. For TCP sockets, `EndPointID` is specified with an IP address, port number pair.

Figure 2 gives psuedo-code that models how client/server applications could directly incorporate RedSocks. The bold code indicates RedSocks specific changes to the typical client/server code. The main responsibility of the client is to handle a new error code, EREDO. EREDO is used to distinguish whether or not the server sends a response to the client in the `redirect` system call. If it does not, the client must "resend" its last request. We present a means to relieve the client application of this responsibility in [10].

3 Implementation

We have implemented RedSocks in the Linux 2.4.16 kernel, which implements BSD sockets that conform to version 4.4BSD.

3.1 Design Issues

The first issue in adding operations on communication end-points, such as redirect, is choosing the OSI model layer to enhance. The four choices are the network, application, transport, and session layers. At the network layer, communication end-points are defined by host addresses. Thus, network layer redirection, such as Mobile IP [14], globally applies to all the host's communication channels. Effectively load-balancing Internet services requires redirecting individual requests, so this type of redirection is often too coarse-grained. Application layer solutions are not transparent To use such a solution, applications must agree on an application protocol, such as HTTP. Also, application layer protocols are more expensive as processing each message requires at least two context switches.[3] Application end-points are available at the transport layer and it would seem a natural extension to perform the redirection operation here. Snoeren et al. [17, 16] use the transport layer to support mobile host migration and fault tolerance for a server cluster. We feel that redirection is a higher level operation that does not coincide with the well defined functions of the transport layer: application end-point addressing, segmentation and assembly, connection control, flow control and error control. In general, end-point operations may require knowledge about sets of communication channels that decisively fall outside the scope of the transport layer. At the session layer, the only objects one has to operate on are communication end-points. According to the OSI, the session layer "is the network *dialog controller*" that "establishes, maintains, and synchronizes the interaction between communicating systems." Sychronization is key in redirecting end-points of communication channels. With this in mind, we created a session layer protocol, EOP, to handle dynamic operations on communication end-points.

[3] A context switch occurs at the arrival of a message, at the departure, and additonal context switches can occur during the protocol processing of the message.

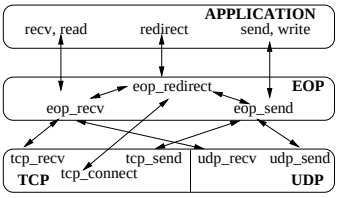


Fig. 3. The architecture for EOP and RedSocks

3.2 Architecture

Figure 3 illustrates our layered architecture. The arrows indicate functional dependencies where data flows through the formal parameters and return values. We defer discussion of the application layer to [10].

End-point Operation Protocol. EOP is a session layer protocol that wraps transport layer data with an EOP header in order to manage end-point movement for the lifetime of the associated communication channel. When end-point movement is required, EOP invokes several functions in the EOP layer on the client and the server to coordinate and accomplish redirection. The EOP header triggers redirection on the remote end-point, synchronizes activity, and exchanges redirection arguments.

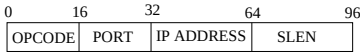
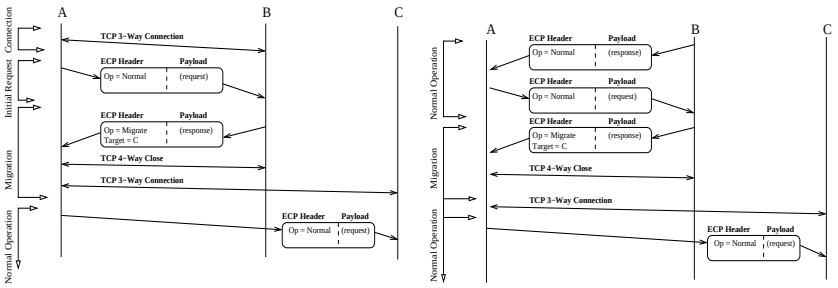


Fig. 4. The EOP Header

We use a simple 12-byte EOP header (see Figure 4) that contains a 16-bit opcode to specify the current operation. The two operations we use are `normal` and `redirect`. To specify the target of a redirect operation we use two parameters: one 16-bit parameter indicating the port number and the other 32-bit parameter indicating the destination IP address. The 32-bit `slen` field records the size of the user message and is used to delineate messages (described further in the transport layer discussion next).

Figure 5(a) shows the time flow diagrams for connection-time and in-stream redirect operations. After connecting to B, A sends its initial request. B uses the `redirect` system call to redirect requests to C. At A, EOP responds to the redirection request by closing the connection to B, opening a new connection to C, and then returning control to the client. Further requests, possibly including the previous request if B did not send a response with its redirect, are sent directly to C. At this point in time, B is fully divorced from A. RedSocks can also be used as an instream redirection mechanism where requests are idempotent or the servers are sharing application state via an alternative method. Figure 5(b)



(a) Connection-time Redirection (b) Instream Redirection

Fig. 5. Time line for EOP packet flow

gives the time flow diagram before, during, and after such a redirection. Its flow is very similar to that depicted by Figure 5(a).

Transport Layer. Linux 2.4.15 provides a well-defined interface for the TCP layer. However, we encountered two difficulties in building our session-layer. The first difficulty is due to the way the network subsystem is optimized. At the session layer, Linux assumes the data to be sent lives in user space, but we need to generate our EOP headers in kernel space. Therefore, we had to modify TCP to handle kernel generated data and use a separate `eop_send` call to deliver it.

Protocol headers encapsulate data packets and convey information on how to process the data in the corresponding layers. However, TCP transparently combines data packets and provides a streaming abstraction of the data. Thus, TCP effectively destroys the association between protocol headers and their data for upstream layers. Our second difficulty is that we needed a way to maintain or rebuild this association. We use the `slen` field in Figure 4 for this purpose. The designers of the Stream Control Transmission Protocol (SCTP) describe this difficulty in RFC2960 and propose SCTP as a solution.

4 Experiments

We ran our overhead experiments (see Section 4.1) on four 400Mhz dual-processor Pentium III machines where each has 256MB of RAM, two 18GB hard drives, and a 100 Mbps Ethernet network card. These machines are connected via a 100 Mbps hub. Each machine ran Linux 2.4.16. In these experiments, data was collected for request/response sizes between 100 bytes and 32K, at 100 byte intervals. Each data point in the graphs represents the average over 100 runs.

For our scalability measurements, we used one client machine, located at the University of California at Davis, of the above type and a small server cluster, located at Vanderbilt University in Tennessee. The server cluster consisted of two

800Mhz AMD machines with 256MB of RAM and 100 Mbps Ethernet network cards. Both campuses are connected via Internet II.

4.1 Overhead Measurements

To better understand the overhead of using RedSocks when redirection is not performed, we measured request-to-response latency from a single client and single server. This effectively measures pure send and receive overheads. For sending and receiving packets, RedSocks incurs a 6.5% average overhead with the Nagle algorithm enabled and a 1.5% average overhead with it disabled.[4]

The behaviour of RedSocks can be mimicked with traditional socket system calls at the application layer. A client can terminate communication with a host by calling `close` on the socket and then re-initiate communication with a different host via calls to `socket` and `connect`. This solution does not require the use of an EOP header, but fails to provide a number of advantages such as client transparency, server-side load balancing control, and dynamic adaptation. Nevertheless, it provides a good baseline against which to compare RedSocks.

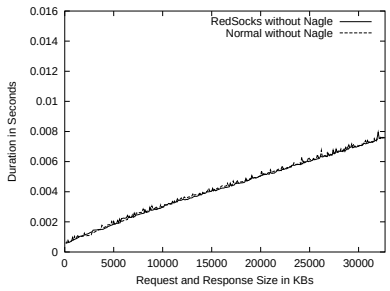


Fig. 6. Overhead of sending and redirection

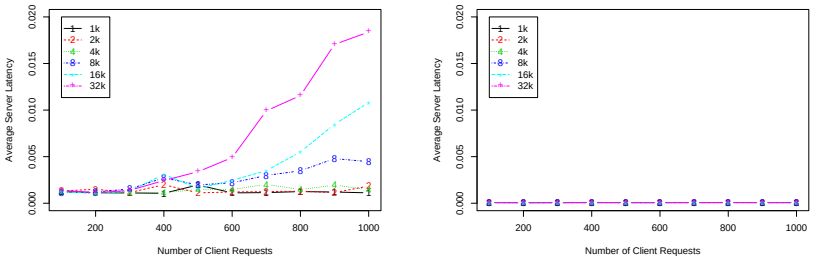
We compare the time it takes to complete two consecutive client requests. With RedSocks, the client sends a request to a host that services the request. In addition, the host redirects its connection end-point to another server that handles the second request. The traditional socket scenario is the same except the client manually reconnects to a host statically specified in the program. We give the transaction time for varying request/response sizes in Figure 6. RedSocks is 0.84% faster when the Nagle algorithm was disabled (see Figure 6).

4.2 Comparing HTTP and RedSocks Redirection latency

In these experiments, the client generates requests to Server 1. Server 1 redirects the requests to Server 2 which handles them. We modified Lynx 2.8.5 to generate user requests in a manner similar to flood[5]. On the servers, we ran either standard

[4] Interested readers can refer to [10] for more details.
[5] http://httpd.apache.org/test/flood/

Apache 1.3.22 or our RedSocks-enabled Apache 1.3.22 depending on whether we were measuring HTTP redirection or RedSocks. We measured server redirection latency when varying the number of requests and file sizes. Server redirection latency is measured from the time the client connects to the time the `redirect`, or HTTP redirection, operation completes. Figure 7 shows a representative subset of these results.



(a) Server Redirection Latency for HTTP redirection for all File Sizes

(b) Server Redirection Latency for RedSocks for all File Sizes

Fig. 7. Server Redirection Latency Measurements

Figure 7(a) gives the average server redirection latency of the Apache web server using HTTP redirection for differing numbers of requests. The results of this graph are counter-intuitive. One would expect that latency would increase when the number of requests are increased. However, observe the latencies for 1k and 2k files. While they are higher than the latencies seen in the corresponding RedSocks results (see Figure 7(b)), they do not increase with the number of requests. Instead, latency *increases with file size*. In analyzing this phenonemon, we discovered an undesirable trait of HTTP redirection: its performance is correlated to the performance of the clients. Thus, a poorly performing client adversely impacts server scalability as witnessed in Figure 7(a).

The explanation of this lies in the steps taken to perform redirection in both mechanisms. For RedSocks, the steps are to (1) `accept` the connection and (2) `redirect` the request. Both of these steps are efficient and neither the file sizes or numbers of requests for our tests effected server redirection latency (see Figure 7(b)). For HTTP redirection, the steps are to (1) `accept` the connection, (2) `read` the request, (3) create an HTTP redirection response, and (4) `send` the response. The time required to generate the HTTP redirection response adds a constant increase to server redirection latency over that incurred by RedSocks. The key to the non-scalable increase in server redirection latency experienced by HTTP redirection is that it must `read` the request (step 2) and `read` is a blocking system call. Usually, the client is responsive and immediately sends its

request resulting in the server having no blocking delay. However, the effects of an unresponsive client can be seen in Figure 7(a) where the client bogs down as file sizes increase. This effect is not specific to HTTP redirection, but occurs in any mechanism that must read a request before redirecting, a common scenario in content-aware redirection.

5 Related Work

DNS can be used to map a single hostname onto multiple hosts [5]. When queried about a name, a modified DNS server returns different IP addresses according to its selection policy. DNS is transparent to both clients and servers making it easy to deploy. It is also convenient because clients must do a DNS lookup to contact Internet services anyway. Unfortunately, DNS-based redirection is coarse-grained, increases load on DNS, and may make poor load-balancing decisions [15]. Several server-based approaches [2, 9] use HTTP redirection, which allows any server to redirect its connections, but is application-specific and has higher overhead. Dispatcher-based mechanisms, such as Cisco's Local Director, Linux's netfilter, TCP splicing [7] and MagicRouter [1], employ network packet rewriting for both connection-time and in-stream connection redirection. Packet rewriting provides a transparent way to build distributed servers, but at a price: it requires a proxy or router to interpose itself between a connection's end-points and actively manipulate packets. In [11], Hunt et al. introduce TCP handoff (also used in LARD [13]), as part of a content-based load distribution scheme, that avoids bi-directional packet rewriting. However, all incoming packets must still pass through a node, which may become a bottleneck. For a more extensive review of related work, please refer to [10].

6 Conclusion

Building scalable, highly-available web servers requires mechanisms that support multiple machines cooperatively handling requests for a service. RedSocks is a new mechanism introduced to solve some of the issues in building distributed architectures that transparently balance load across servers. With RedSocks, one can manipulate the end-points of the communication. This solution improves upon previous ad hoc mechanisms for balancing load among servers in that it is scalable, fine-grained, and transparent to the client. In future work, we will explore extensions, such as support for fault tolerance and lazy redirection, which allows a server to redirect a connection that was just redirected to it.

References

1. E. Anderson, D. Patterson, and E. Brewer. The MagicRouter: An application of fast packet interposing. `http://www.cs.berkeley.edu/~eanders/projects/magicrouter/`, October 1996.

2. D. Andresen, T. Yang, O. Ibarra, and O. Egecioglu. Adaptive partitioning and scheduling for enhancing WWW applications performance. *Journal of Parallel and Distributed Computing*, 49(1):57–85, February 1998.
3. M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Proc. USENIX 2000 Annual Technical Conference*, San Diego, CA, USA, 18–23 June 2000.
4. N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *Proceedings of the Ninth International World Wide Web Conference*, volume 33(1–6) of *Computer Networks*, pages 1–16, Amsterdam, The Netherlands, 15–19 May 2000.
5. T. Brisco. DNS support for load balancing. RFC 1794, Rutgers University, April 1995.
6. V. Cardellini, M. Colajanni, and P. Yu. Dynamic load balancing on web-server systems. In *IEEE Internet Computing*, pages 28–39. IEEE, May-June 1999.
7. A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. In *Proc. 2nd USENIX Symposium on Internet Technologies and Systems*, pages 117–26, Boulder, CO, USA, 11–14 October 1999.
8. O.P. Damani, P.E. Chung, Y. Huang, C. Kintala, and Y. Wang. ONE–IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems; Sixth International WWW Conference*, 29(8–13):1019–27, 7–11 April 1997.
9. M. Garland, S. Grassia, R. Monroe, and S. Puri. Implementing distributed server groups for the world wide web. Technical report, Carnegie Mellon University, January 1995.
10. Michael Haungs, Raju Pandey, Earl Barr, and J. Fritz Barnes. A fast connection-time redirection mechanism for internet application scalability. Technical Report CSE-2001-10, University of California, Davis, March 2001.
11. G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.
12. *the Proceedings of IEEE INFOCOM 2001*, Anchorage, AK, USA, April 2001.
13. V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 205–216, San Jose, California, 1998.
14. C. Perkins. IP mobility support. Internet Request for comments (RFC 2002), October 1996.
15. A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of DNS-based server selection. In INFOCOM2001 [12], pages 1801–10.
16. A. Snoeren, D. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 221–232, San Francisco, CA, March 2001.
17. A.C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of the 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 155–164, August 2000.