# Fine-Grained Failover Using Connection Migration

Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan
*MIT Laboratory for Computer Science*
*Cambridge, MA 02139*
{snoeren, dga, hari}@lcs.mit.edu

## Abstract

This paper presents a set of techniques for providing fine-grained failover of long-running connections across a distributed collection of replica servers, and is especially useful for fault-tolerant and load-balanced delivery of streaming media and telephony sessions. Our system achieves connection-level failover across both local- and wide-area server replication, without requiring a front-end transport- or application-layer switch. Our approach uses recently proposed end-to-end "connection migration" mechanisms for transport protocols such as TCP, combined with a soft-state session synchronization protocol between replica servers.

The end result is a robust, fast, and fine-grained connection failover mechanism that is transparent to client applications, and largely transparent to the server applications. We describe the details of our design and Linux implementation, as well as experimental data that suggests this approach is an attractive way to engineer robust systems for distributing long-running streams; connections suffer relatively small performance degradation even when migration occurs every few seconds, and the associated server overhead is small.

## 1 Introduction

Ensuring a high degree of reliability and availability to an increasingly large client population is a significant challenge facing Internet content providers and server operators today. It is widely recognized that the computers serving Web content and streaming media on the Internet today do not possess the same impressive degree of reliability as other mission-critical services such as gateways and switches in the telephone network.

An effective way to engineer a reliable system out of unreliable components is to use redundancy in some form, and server replication is the way in which reliable and available services are provided on the Web today. Because most Web connections last for relatively short periods of time, the problems of load-balancing client requests and recovering from unreachable replica servers can usually be handled at the granularity of a complete connection. Indeed, this is the approach seen in several current systems that perform server selection using a front-end transport- or application-layer switch [2, 6, 11, 18, 20], or using wide-area replication for Web content distribution [1, 8].

While existing replication technologies provide adequate degrees of reliability for relatively short Web connections, streaming media and Internet telephony display substantially longer transfer lengths. Providing reliable, robust service over long connections requires the ability to rapidly transition the client to a new server from an unresponsive, overloaded, or failed server *during* a connection [16]. The requirements of these emerging applications motivate our work.

We have designed and implemented a system that achieves fine-grained, connection-level failover across both local- and wide-area server replicas, *without* a front-end transport- or application-layer switch. Thus, there are no single points of failure or potential front-end bottlenecks in our architecture. We achieve this using a soft-state session synchronization protocol between the replica servers, combined with a connection resumption mechanism enabled by recently-developed end-to-end *connection migration* mechanisms for transport protocols such as TCP [22] and SCTP [25]. The end result is a robust, fast, and fine-grained server failover mechanism that is transparent to the client application, and largely independent of the server applications. Applications that can benefit from this include long-running TCP connections (e.g., HTTP, FTP transfers), Internet telephony, and streaming media, enabling them to achieve "mid-stream failover" functionality.

Our architecture is end-to-end, with active participation by the transport stack of all parties involved in the communication. Our design is largely application-

This research was supported in part by DARPA, NSF, NTT Corporation, Intel, and IBM. Alex C. Snoeren is supported by a National Defense Science and Engineering Graduate (NDSEG) Fellowship.

independent: applications do not need to be modified to benefit from the fine-grained failover techniques. However, because we allow a server to seamlessly take over a connection from another in the middle of a data stream, there needs to be some mechanism by which the servers synchronize application-level state between themselves. While this is a hard problem in general, there are many important common cases where it is not as difficult. For example, when each client request maps directly to data from a file, our lightweight synchronization mechanism performs quite well. If content is being generated dynamically in a fashion that is not easily reproduced by another server, handoff becomes harder to accomplish without additional machinery.

We discuss several issues involved in designing a connection failover mechanism in section 2. Section 3 describes an end-to-end architecture for fine-grained server failover targeted at long-running transfers. Our TCP-based Linux implementation is described in section 4. Section 5 contains performance studies showing the effectiveness of the failover mechanism and its resilience to imperfections in the health monitoring subsystem. We conclude with a synopsis of related work in section 6 and summarize our contributions in section 7.

# 2 Components of a failover system

This section describes three components that a complete fine-grained failover system should provide:

(i) For any connection in progress, a method to determine when (and if) to move it to another server;

(ii) a selection process to identify a set of new server candidates; and

(iii) a mechanism to move the connection and seamlessly resume the data transfer from the new server.

## 2.1 Health monitoring

In general, the end-point of a connection is changed because the current server is unresponsive; this may happen because the server is overloaded, has failed, or its path to the client has become congested. The failover system needs to detect this, following which a new server can be selected and the connection appropriately moved. We call the agent in the failover system that monitors the health and load of the servers the *health monitor*.

There are several possible designs for a health monitor, and they can be broadly classified into centralized and distributed implementations. Our architecture accommodates both kinds. We believe, however, that it is better for the health monitor to be controlled by the servers than the client. It is often much harder for a client to have the requisite knowledge of the load on other servers, and putting it in control of making movement decisions may actually worsen the overall performance of the system.

The focus of this paper is not on novel mechanisms for load or health monitoring; instead, we leverage related work that has already been done in this area [3, 12, 17]. For the purposes of this paper, we assume that each server in a support group is notified of server failure by an omniscient monitor at the same time. As the experimental results in section 5 demonstrate, however, our system allows the health monitoring component to be relatively simple—*and even overly reactive*—without significantly degrading performance.

## 2.2 Server selection

Once the system decides that a connection should be moved to another server, it must select a new server to handle the connection. One possibility is to use a content routing system and treat this as a new request to decide which server to hand it to. Another is to have the set of relatively unloaded servers attempt to take over the connection, and arrange for exactly one of them (ideally the closest one) to succeed. Our failover architecture admits both styles of server selection.

## 2.3 Connection migration and resumption

Once a connection has been targeted for movement and a new server has been selected to be the new end point, the client application should seamlessly continue without noticeable disruption or incorrect behavior. This requires that the application data stream resume from exactly where it left off at the old server. To achieve this in an application-independent fashion, the transport-layer state must be moved to the new server, and the application-layer state appropriately synchronized and restarted.

There are different ways of doing this: one is a mechanism integrated with the application where the clients and servers implement a protocol that allows the server to inform the client that its communicating peer will change to a new one. Then, the client application can terminate the connections to the current server and initiate them to the new one, and retrieve the portions of the stream starting from where the previous server left off. An alternate approach, which our work enables and advocates, is an application-independent mechanism by using a secure, transport-layer connection migration mechanism. The advantage of this method is that existing applications can benefit from this without modification, while new ones do not each need to incorporate their own mechanisms to achieve these results.

# 3 Architecture

Our architecture preserves the end-to-end semantics of a connection across moves between servers. Rather than inserting a Web switch or similar redirecting device, we associate each connection with a subset of the servers in the system[1]. This is the connection's *support group*, the collection of servers that are collectively responsible for the correct operation of the connection. Each support group uses a soft-state synchronization protocol to distribute weakly consistent information about the connection to each server in the group. This information allows a stream to resume from the correct offset after a move.

When the health monitor determines that a connection should be moved, each of the remaining servers in the connection's support group becomes a *candidate server* for the orphaned client connection. Thus, the responsibility for providing back-up services to orphaned connections is shared equally among the other servers, and the new server is chosen from the support group in a decentralized manner by the client.

Determining the precise point of failure of a server is a difficult problem, but is fortunately one that does not need to be solved. It is sufficient to determine when the client believes the server failed. In the case of sequenced byte streams, this can be represented by the last successfully received contiguous byte as conveyed in the transport-layer acknowledgment message. Hence the new server need only elicit an acknowledgement packet from the client in order to determine the appropriate point from which to resume transmission.

The state distribution protocol periodically disseminates, for each connection, the mapping between the transport layer state and the application-level object being sent to the client (e.g., the TCP sequence number and an HTTP URL). For the remainder of this paper, we will use the term *stream mapping* to describe the task of translating the particular transport-layer sequence information into application-level object references the server can understand. The transport-layer state is moved to the new server using a secure *connection migration* mechanism [22]. Together, the techniques described above allow for the correct resynchronization of both transport-layer and application state, transparent to the client application

Figure 1 shows the basic architecture of our system for a simple configuration with two servers, $A$ and $B$, in a support group. When a client wishes to retrieve an object, it is initially directed to Server $A$ through some server selection mechanism. The connection is observed by the stream mapper, which parses the initial object request and advertises the object currently being served and the necessary stream mapping information to the rest of the support group.

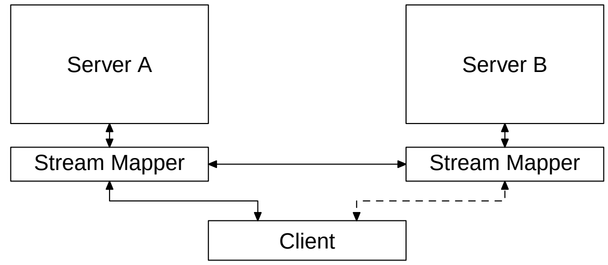


**Figure 1: Failover architecture for a support group of two servers.**

At some point Server $B$ may attempt to assume control of the connection. This may be caused by the receipt of a notification from the health monitor that Server $A$ died, or may be initiated by a load-balancing policy mechanism. Server $B$ initiates a connection migration by contacting the client *in-band* on the same connection, using a secure transport-layer connection migration mechanism. If it receives a transport-layer acknowledgement from the client, it knows that it was the candidate server selected by the client. Because this acknowledgment notifies the server of the next expected data byte, and because the soft-state synchronization protocol in the support group has already informed Server $B$ of the content being served, content delivery can now be resumed at the correct point. Observe that at no point was the client actively involved in the decision to migrate the connection, yet its transport-layer is fully aware that the migration took place and it even had the opportunity to pick one of several candidate servers[2]. All further client requests on the same connection are now directed to Server $B$, at least until the next migration event.

In the rest of this section, we describe the formation and maintenance of support groups, the soft-state synchronization protocol, and the details of the transport-layer connection migration.

## 3.1 Support groups

The size and distribution of support groups clearly has a large impact on the performance of our scheme. As the number of servers in a support group increases, the increased load to any one server of a death in the group decreases. Unfortunately, the communication load also

[1] This subset could in fact be the entire set; partitioning the set into subsets enhances scalability.

[2] If required, it is possible to provide information about the migration to the client application; we have not yet explored the ramifications of this, but plan to do so in the future.

increases, as each member of the group must advertise connection state to the others.

Since the set of candidate servers for a client whose initial server dies is bounded by the set of servers in the support group, it may be desirable to have a diversity of network locations represented in the support group to increase the chance the client has an efficient path to a candidate server. This depends greatly, however, on the effectiveness of the initial server selection mechanism, and the stability of that choice over the duration of the connection. We note that it may be desirable to limit the number of candidate servers that simultaneously attempt to contact the client in large support groups, as the implosion of migration requests may swamp the client. As a practical matter, the quadratic growth in communication required between the servers in a support group will likely limit support group size to a reasonable number.

Clearly, the choice of a live initial server is an important one, and much previous work has addressed methods to select appropriate servers in the wide area. Good servers can be identified using BGP metrics [5], network maps [1], or application-specific metrics. Similarly, clients can be directed to these servers using DNS redirection [1], HTTP redirection, BGP advertisements, or anycast [9]. Since our architecture allows connections to be handed off to any other server at any point in the connection, the ramifications of a poor initial selection are not as severe as with current schemes.

We construct support groups by generating a well-known hash of the server's IP address. By setting the number of hash buckets to $n$, servers are uniformly allocated into one of $n$ support groups. This mechanism allows for the servers to be dynamically added and removed from the server pool. As each server comes on-line, it computes the hash of its interface address and then begins advertising its connections to a well-known multicast address for that hash value[3]. Every server in the support group becomes a candidate server for the connections from the new server immediately after receiving the first advertisement. Similarly, the new server begins listening to advertisements sent to the group's address, and becomes a candidate server for any connections advertised after its group membership[4].

The choice of support group membership and final server that handles a failed client should be engineered in a manner that avoids the "server implosion" problem, where all of the clients from a failed server converge on the same replacement server. We provide two mechanisms to support implosion avoidance. The first is an engineering choice: by choosing smaller support group sizes ($g$ members per support group out of $n$ total servers), we can limit with high probability the expected number of clients that converge on a particular server to an $O(g/n)$ fraction of the clients from the dead server. The second is by delaying connection resumption by a load-dependent factor at each server, increasing the likelihood the client is served by a less crowded server. We evaluate this technique in section 5.

### 3.2 Soft-state synchronization

The information advertised to the support group about each content stream can be divided into two parts. The first portion contains application-dependent information about the object being retrieved from the server, such as the HTTP URL of the client request. The second portion is the transport-layer information necessary to migrate the connection. In general, this includes the client's IP address, port number, and some protocol-specific transport layer state, such as the initial sequence number of the connection. The amount and type of transport-specific information varies from protocol to protocol. We are currently exploring a framework that describes the necessary information in a protocol-independent fashion.

The state information for a particular connection may be unavailable at some servers in the support group because the connection has not yet been announced, or because all of the periodic announcements were lost. The first failure mode affects only a small window of new connections, and can be masked as an initial connection establishment failure. We assume that connections are long-running, so the second failure mode must be guarded against more carefully. If at least one server in the support group has information about the connection, it can be re-established. We therefore need only bound the probability that no servers have fresh information about connection. It suffices to pick suitable advertisement frequencies and support group sizes (see, e.g., the analysis of a similar protocol by Raman *et al.* [19]), hence it is possible to achieve sufficient robustness in our synchronization protocol with markedly less complexity than a strongly-consistent mechanism.

### 3.3 Transport-layer connection migration

When attempting to resume a connection previously handled by another server in a fashion transparent to the client application, previous approaches have forged the server's IP address, making packets from the new server indistinguishable from the previous ones. This approach has two major drawbacks:

[3] The required multicast functionality may be provided at the IP layer or through an application-layer overlay.

[4] The initial request redirection mechanism must also be informed of the new server if it is to handle new client requests.

- The new server and the failed server must be co-located. Since they both use the same IP address, packets from the client will be routed to the same point in the network.
- The previous server cannot be allowed to return to the network at the same IP address, for otherwise there will be two hosts with the same address. Worse, if the initial server attempts to continue serving the connection, confusion may ensue at the client's transport layer.

Current approaches take advantage of the first requirement to ameliorate the second. Since both the initial server and the failover server must be co-located, so-called layer-four switches or "Web switches" are placed in front of server groups. Web switches multiplex incoming requests across the servers, and rewrite addresses as packets pass through. This enables multiple servers to appear to the external network as one machine, providing client transparency. The obvious drawback of this approach, however, is that all servers share fate with the switch, which may become a performance bottleneck.

By using explicit connection migration, which exposes the change in server to the client, we remove both of these restrictions. (Indeed, our approach further empowers the client to take part in the selection of the new server.) Servers can now be replicated across the wide-area, and there is no requirement for a redirecting device on the path between client and server.[5]

Explicitly informing the client of the change in server provides robust behavior in the face of server resurrection as well. If the previous server attempts any further unsolicited communication with the client—possibly due to network healing, server restart, or even a false death report by the health monitoring system, the client simply rejects the communication in the same fashion as any other unsolicited network packet.

Our architecture leverages the absence of the co-location requirement by allowing the client to select its candidate server of choice. This can be done in the transport layer by simply accepting the first migration message to arrive. Assuming that all servers in the support group are notified of the current server's failure (section 2.1), the first request to arrive at the client is likely from the server best equipped to handle the message. The response time of a candidate server is the sum of the delay at the server and propagation delay of the request to the client. While not guaranteed to be the case due potentially unstable network conditions in the Internet, in general a candidate server with good (low latency) connectivity and low load is likely to win out over a loaded candidate server with a poor route to the client.

Clearly there are exceptions to this rule, but we believe that it is a reasonable starting point. We note that if a more sophisticated decision process is desired it can be implemented either at the candidate servers, since each is aware of not only the client to be contacted, but has a reasonable knowledge of the identities of the other members of the support group, or the client application, or both.

### 3.3.1 Protocol requirements for migration

Our architecture fundamentally requires the ability to migrate transport-level connections. While not widely available today, we believe that this capability is a powerful way to support both load-balancing and host mobility, and can be deployed incrementally by backward-compatible extensions to protocols like TCP [22] and as an inherent feature of new protocols like the Stream Control Transport Protocol (SCTP) [25].

In addition to basic address-change negotiation, a migrateable transport protocol must provide two features to support our failover architecture: (i) it should be possible for the migration to be securely initiated by a different end-point from the ones used to establish the connection, and (ii) the transport mechanism must provide a method for extracting the sequence information of the last successfully received data at the receiver, so that the resumption can proceed correctly. Furthermore, the performance of our scheme is enhanced by the ability of multiple candidate servers to simultaneously attempt to migrate the connection, with the guarantee that at most one of them will succeed. We now address these issues in the context of TCP and SCTP.

### 3.3.2 TCP migration

While not a standard feature of TCP, extensions have been proposed to support connection migration, including the recently-proposed TCP Migrate Options [22, 23]. Using these options, correspondent hosts can preserve TCP connections across address changes by establishing a shared, local connection token for each connection. Either peer can negotiate migration to a new address by sending a special Migrate SYN segment containing the token of the previous connection from the new address. A migrating host does not need to know the IP addresses of its new attachment point(s) in advance.

Each migration request includes a sequence number, and any requests with duplicate sequence numbers are ignored (actually, they are explicitly rejected through a RST segment). This provides our scheme with the needed at-most-once semantics—each candidate server

[5] The stream mapper is not such a device; rather, it is a software module that allows a server application to participate in the soft-state synchronization protocol and connection resumption mechanism.

sends a Migrate Request with the same sequence number; the first packet to be received at the client "wins," and the rest are rejected. Furthermore, once the connection is migrated, packets from the previous address are similarly rejected, hence any attempt by the previous server to service the client (perhaps due to network healing or a erroneous death report) are actively denied.

The Migrate SYN as originally specified used the sequence number of the last transmitted data segment. When packets are lost immediately preceding migration, retransmissions from the new address carry a sequence number earlier than the Migrate SYN. Unfortunately, several currently-deployed stateful firewalls block these seemingly spurious data segments, considering them to be a security risk.

We remedy the situation by modifying the semantics of the Migrate SYN to include the sequence number of the last data segment successfully acknowledged by the client. This ensures that all data segments transmitted from the new address will be sequenced after the Migrate SYN. Further, since a host cannot reliably know what the last successfully acknowledged segment number is (since ACKs may be lost), we relax the enforcement of sequence number checking on Migrate SYNs.

By allowing the Migrate SYN to fall outside of the current sequence space window, however, an attacker does not need to know the current sequence space of a connection to hijack it. Hence, in our extended model, only the secured variant of the Migrate Options provides protection against hijacking by an eavesdropper. The secured form of the Migrate Options uses an Elliptic Curve Diffie-Hellman key exchange during the initial three-way handshake to negotiate a cryptographically-secure connection key. Secure Migrate SYNs must be cryptographically authenticated using this key, hence an attacker without knowledge of the connection key cannot hijack a connection regardless of the current sequence space.

This relaxation allows any server with sufficient knowledge of the initial transport state (namely the initial sequence number, connection token, and key) to request a migration. When a server with stale transport layer state assumes control of a connection, however, the client needs to flush its SACK blocks (and corresponding out-of-order packets) for proper operation of the TCP stack at the new server.[6] Otherwise, the transmission of a data segment that fills a gap in previously-transmitted out-of-order segments will cause the client to acknowledge receipt of data that the new server has not yet sent, confounding the server's TCP (and an untold number of middle boxes).

In general, a host cannot deduce the difference between a Migrate SYN issued by the original host (from a new address) that simply failed to receive some number of ACKs and a new end point. We therefore reserve one bit from the Migrate SYN to flag when a Migrate Request is coming from a host other than the one that initiated the connection [23]. This allows TCP connections migrated by the same host to avoid the negative performance implications of discarding out-of-order segments while providing correct operation in the face of end host changeover.

### 3.3.3 SCTP migration

Recent work in IP telephony signaling has motivated the development of a new transport protocol by the transport area of the IETF (Internet Engineering Task Force). A proposed standard, the Stream Control Transport Protocol (SCTP), provides advantages over TCP for connections to multi-homed hosts [25]. By advertising a set of IP addresses during connection establishment, a multi-homed SCTP connection supports transmitting and receiving data on multiple interfaces. We can leverage this capability to support connection migration between different servers.

Unlike the TCP Migrate options, however, all addresses to be used for an SCTP connection must be specified at connection establishment. While limiting the level of dynamism in the server pool, it still supports failover between servers that were known to the initial server at the time of connection establishment. A recent Internet Draft [26] attempts to address this issue by allowing for the dynamic addition and deletion of IP addresses from the association, although it requires the operations to be initiated by an end point already within the association. Additionally, SCTP was designed to support multi-homed hosts, as opposed to address changes, hence it does not support the at-most-once semantics required to allow multiple servers to simultaneously attempt to migrate the connection. We believe this is a deficiency in SCTP that can be addressed simultaneously while adding support for dynamic changes to the address associations.

The additional complexity of SCTP requires servers to communicate more transport-level state information, but need not increase the frequency of communication. SCTP end-points are required to emit SACK packets upon receipt of duplicate data segments, hence the new server could send a data segment with a Transport Sequence Number (TSN) that is known to be stale, eliciting a SACK with the current sequence state.

[6] This behavior is in full compliance with the SACK specification [15].

### 3.4 Limitations

Our architecture depends on the ability to perform the stream mapping function for objects being requested. Due to variability in header lengths, this requires access to the transport layer immediately below the application. In many cases, there is only one (non-trivial) transport protocol in use, such as TCP or SCTP. In some instances, however, transport protocols may be layered on top of each other, such as RTP over TCP. In this instance, both migration and stream mapping must be performed at the highest level, namely RTP.

Independent of the transport- and network-layer issues handled by our architecture, particular applications may attempt to enforce semantics that are violated by a server change; clearly such applications cannot be handled in a transparent fashion. In particular, the object being served may have changed since the connection was initially opened, resulting in indeterminate behavior if the application isn't aware of this. Further, some applications make decisions at connection establishment based on server- or time-sensitive state, and do not normally continue to reevaluate these conditions. For instance, current applications may make authorization decisions based on the IP address of its original peer, or an HTTP cookie or client certificate that has since expired. Such applications can be notified of the connection migration by the stream mapper, however, and allowed to perform any required steps to resynchronize themselves before resuming transmission.

Despite these limitations, the common case of a long-running uni-directional download from a static file or consistent stream is handled by our architecture in an application-transparent manner. As described in the following sections, our implementation allows a connection to be migrated at any point after the initial object request completes. Furthermore, our scheme can survive cascaded failures (when the back-up server fails before completing the connection migration) due to the semantics of our state distribution mechanism and the robustness of the transport-layer migration functionality.

## 4 Implementation

Our current prototype implementation supports TCP applications using the Migrate options [22, 23]. The soft-state distribution mechanism and stream mapping functionality are implemented as a *wedge* that runs on the server and proxies connections for the local content server. Our current implementation uses Apache 1.3, and is compatible with any Web server software that supports HTTP/1.1 range requests. It is possible to optimize performance by handing off the TCP relaying to the kernel after the request parameters have been determined, in a manner akin to MSOCKS [14].

### 4.1 The wedge

The wedge is a TCP relay that accepts inbound connections from clients, and forwards them to the local content server. It listens in on the initial portion of a connection to identify the object being requested by the client, examines the returned object to determine the parameters necessary to resume the connection if necessary, and then hands the relaying off to a generic TCP *splice* to pass the data between the client and the server. The use of a user-level splice to copy data from one TCP connection (server-to-wedge) to another (wedge-to-client) results in some performance degradation compared to an in-server or in-kernel implementation, but permits a simple and clean implementation that can be used with a variety of back-end servers, not just one modified server.

Each application protocol handled by the wedge requires a parsing module to identify the requested object and strip out any protocol headers on the resumed connection. The architecture of the wedge in the context of HTTP is shown in figure 2.

The wedge first passes the connection through the HTTP GET parser, which watches the connection for a GET request to identify the requested object. The parser passes the request along to the back-end server, and hands off further control of the connection to the HTTP header parser. The parser counts the length of the HTTP headers to identify the offset in the data stream of the beginning of the object data, and forwards the headers back to the client. It then passes control to the generic data relay, which maintains control of the connection until termination.

Finally, a protocol-independent soft-state distributor periodically examines the set of active connections and sends information about each connection (over UDP) to its support group. This information includes:

- Client IP address and port number.
- Takeover sequence number.
- TCP Migrate fields (connection token, key, etc.).
- Application-specific object parameters, including the name of the object.

Each server maintains a table of all the connections in its support group(s). When a server receives notification that a peer has failed, it determines if it is in the support group for any connections previously managed by the dead server (using the hashing mechanism described in section 3.1), and attempts to resume the connections.

To do so, it migrates the client connection (section 4.2),

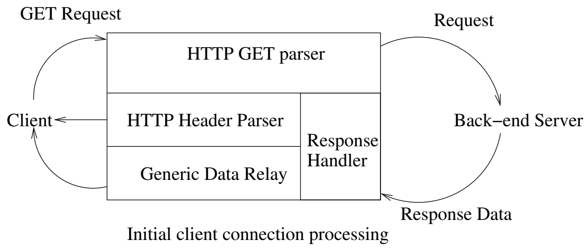


**Figure 2: The wedge handling a new connection (left) and taking over an existing connection (right).**

computes the new offset into the data stream by comparing the current TCP sequence number to the connection's initial sequence number, and then passes the client connection to the protocol re-establishment module to continue the connection. For HTTP, the re-establishment module sends an HTTP range request to the local server, strips out the protocol headers, and then relays the data to the client, seamlessly resuming the transfer.

To avoid race conditions, connection migrations are sequenced. If a server hears an announcement for a connection it believes it is currently managing, it checks the sequence number. If that number is greater than its own, it can safely infer that the connection has been migrated elsewhere (due to a load-balancing policy decision or an erroneous death report), and it may terminate its connection and stop advertising it to others. Similarly, peers accept only the most recent announcement for a connection, facilitating convergence after a migration.

#### 4.1.1 Soft-state information dissemination

Soft-state information updates are sent every `UPDATE_SEC` seconds, and expired from remote servers after `CACHE_TIMEOUT` seconds. The proper values for these parameters depend on several factors [19]: connection lengths, packet loss rates between servers, and connection frequency. Our default values for these are an update frequency of three seconds and a timeout period of 10 seconds, which result in acceptable update frequencies when serving typical streaming media connections, without undue state dissemination overhead. The optimal values for these figures would, of course, vary from site to site.

#### 4.1.2 Persistent connections

Our wedge implementation does not currently support persistent connections, but could do so with only minor modification. The wedge would need to continually monitor the client side of the connection (instead of blindly handing the connection off to a splice) and watch for further object requests. When it receives a new request, the wedge would begin announcing it via the soft-state distribution protocol. Using techniques from the LARD persistent connection handler [4], the splice can still perform fast relaying of the bulk data from the server to the client. An in-server implementation of the soft-state dissemination protocol would of course eliminate this need.

### 4.2 Socket interface

In order for a new server to handle a migrated connection, the wedge must preload a socket with sufficient transport information. Conversely, this information must be extracted from the socket at the previous server and communicated to the new server. We have implemented two new system calls, *setsockstate()* and *getsockstate()*, to provide this functionality.

The *getsockstate()* call packages up the TCP control block of an existing TCP socket, while the *setsockstate()* call injects this information into an unconnected TCP socket. Only certain parts of the control block are relevant, however, such as the sequence space information, TCP options, and any user-specified options such as an MTU clamp. Other, unportable state such as the retransmission queue, timers, and congestion window are returned by *getsockstate()*, but not installed into the new socket by *setsockstate()*. Invoking the *connect()* system call will then cause the socket to attempt to migrate the connection (as specified by the preloaded state) to the new socket.

After migration is completed (the *connect()* call succeeds), the new server can compare the current sequence number to the initial sequence number returned by *getsockstate()* to determine how many bytes have been sent on the connection since its establishment.

### 4.3 Migration

Figure 3 presents a `tcpdump` trace of a failover event, collected at the client. There are four hosts in this example: the client, cl, and three servers, sA, sB, and sC. To simulate a diverse set of realistic network conditions, the servers are routed over distinct DummyNet [21] pipes with round trip times of approximately 10ms, 40ms, and 200ms respectively. Each pipe has a bottleneck bandwidth of 128Kb per second. Initially the client is retrieving an object from sA. After some period of time, an

**Initial Data Transmission:**

```
 0.00000 cl.1065 > sA.8080:  .  ack 0505 win 31856
```

(Erroneous) *sA* Death Pronouncement Issued

```
 0.08014 sA.8080 > cl.1065:  P 0505:1953(1448) ack 1 win 31856
```

**Successful Connection Migration to sB:**

```
 0.09515 sB.1033 > cl.1065:  S 0:0(0) win 0 <migrate PRELOAD
1>
 0.09583 cl.1065 > sB.1033:  S 0:0(0) ack 1953 win 32120
 0.14244 sB.1033 > cl.1065:  .  ack 1 win 32120
```

**Continued Data Transmission from sA:**

```
 0.17370 sA.8080 > cl.1065:  P 0505:1953(1448) ack 1 win 31856
 0.17376 cl.1065 > sA.8080:  R 1:1(0) win 0
```

**Failed Connection Migration Attempt by sC:**

```
 0.17423 sC.1499 > cl.1065:  S 0:0(0) win 0 <migrate PRELOAD
1>
 0.17450 cl.1065 > sC.1499:  R 0:0(0) ack 1 win 0
```

**Resumed Data Transmission from sB:**

```
 0.24073 sB.1033 > cl.1065:  P 1953:3413(1460) ack 1 win 32120
 0.25663 cl.1065 > sB.1033:  .  ack 3413 win 31856
 0.33430 sB.1033 > cl.1065:  P 3413:4873(1460) ack 1 win 32120
 0.42776 sB.1033 > cl.1065:  P 4873:6333(1460) ack 1 win 32120
 0.42784 cl.1065 > sB.1033:  .  ack 6333 win 31856
 .
 .
 .
```

**Figure 3: An annotated failover trace (collected at the client) depicting the migration of a connection to one of two candidate servers.**

erroneous death pronouncement is simultaneously delivered to the servers by a simulated health monitor that declares sA dead. This pronouncement is received by the servers at approximately 0.073s (not shown).

As figure 3 shows, each of the other servers in the support group immediately attempts to migrate the connection. Due to their disparate path latencies, the Migrate SYNs arrive at different times. With a path latency of only 20ms, the Migrate SYN from sB arrives first, and is accepted by the client.

The next section of the trace shows the robustness of our scheme in the face of mistaken death pronouncements. The previous simulated announcement by the health monitor was in error; sA is in fact still operational. Furthermore, there are several outstanding unacknowledged data segments (including the segment seen in the trace at time 0.08014), as the client does not emit any further ACKs to sA once it has migrated to sB. Hence sA times out and retransmits the most recent data segment. The client responds by sending a RST segment, informing sA it is no longer interested in receiving further transmissions. (The remaining retransmissions and corresponding RSTs are not shown for clarity.)

Continuing on, we see that the Migrate SYN from the other candidate server, sC finally arrives approximately 100ms after the death announcement. Since the Migrate Request number (1) is identical to the previously received Migrate SYN, the client rejects the request. Note that if sB had died, and this Migrate SYN was an attempt by sC to further resume the connection, the request number would have been incremented.

The final portion of the trace shows the resumed data transmission, continuing from the last contiguous received data segment (as indicated by the SYN/ACK sent by the client). Since the path from client to sB is likely different from the path to sA, the TCP congestion state is reset and the connection proceeds in slow start.

## 5 Performance

We conducted several experiments to study the robustness of our scheme in the presence of overly-reactive or ill-behaving health monitors, the overhead incurred, and the consequences of many connections requiring simultaneous migration.

### 5.1 Server stability

We first examined the performance degradation experienced by a connection as a function of the rate at which it is migrated between different servers. The lower the impact, the greater the resilience of our scheme to an imperfect health monitor or unstable load-balancing policy. In particular, we would like to isolate the highest migration frequency before performance severely degrades.

One might assume that performance degradation would increase steadily as the frequency of oscillations between servers increases. Hence we conducted a series of simple experiments where a client was connected to two servers over distinct links, each with a round-trip propagation time of 40ms and distinct bottleneck bandwidths of 128Kbps. The bottleneck queue size from both servers was 14 KBytes, substantially larger than the bandwidth-delay product of the path. All graphs in this section represent data collected at the client.

Contrary to our initial intuition, we find that the degradation is non-monotonic in the oscillation frequency for this experiment. This is shown in figure 4, which depicts the progression of five separate downloads, each subjected to a different rate of oscillation. The connection served entirely by one server performs best, but the other rates deviate in an unexpected manner.

While the traces and exact numbers we present are specific to our link parameters, they illustrate three important interactions. The first is intuitive: the longer the time between server change events, the higher the throughput, because there is less disruption. This explains the decreasing overall trend and the decreasing magnitude of the "bumps" in figure 5. The second effect is due to the window growth during slow start; if migrations occur before the link bandwidth is fully utilized, throughput decreases dramatically because the connection al-

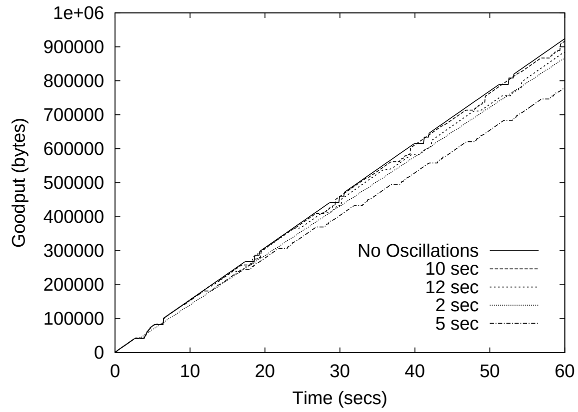


**Figure 4: Connection ACK traces for varying rates of server oscillation.**

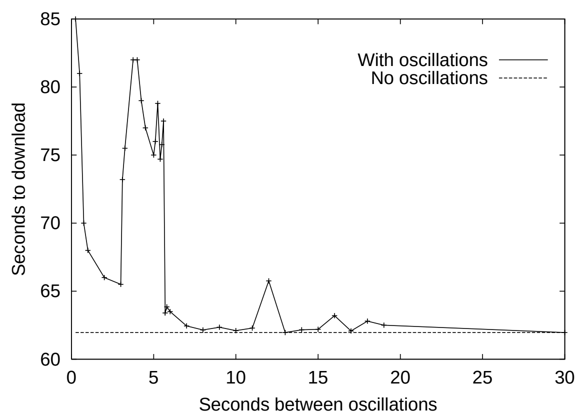


**Figure 5: Download times vs. oscillation rates. A connection served entirely by one server takes 61.96 seconds to complete.**

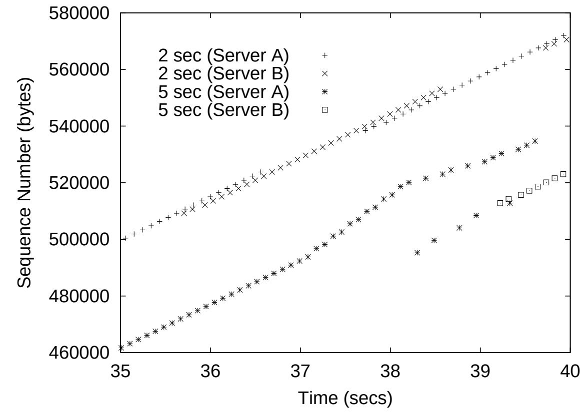


**Figure 6: Sequence traces of oscillatory migration behavior.**

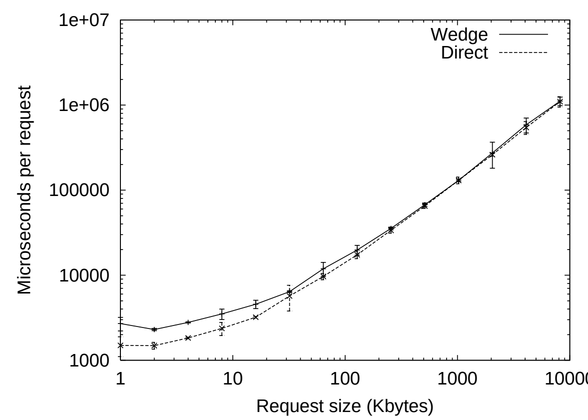


**Figure 7: The request overhead of the wedge as a function of request size.**

ways under-utilizes the link. This occurs at at oscillation periods less than three seconds in the figure 5. The third interaction occurs when migration occurs during TCP loss recovery, either due to slow start or congestion avoidance. In this case, the *go-back-n* retransmission policy during migration forces the connection to discard already-received data. This interaction explains the periodic "bumps" in figure 5 and is discussed in more detail below.

To illustrate the slow-start and loss recovery interactions, figure 6 examines the sequence traces for the interval from 35 to 40 seconds of the connections subjected to 2 and 5 second oscillations. At 2 seconds, connections are still ramping up their window sizes and have experienced no losses. At 5 seconds, the connections experience multiple loss events as slow start begins to overrun the buffer at the bottleneck. Four retransmissions can be observed to be successfully received, the fifth unfortunately arrives *just after* the Migrate SYN from the new server. Since the remaining data is non-contiguous, it is flushed at the receiver in accordance with the *go-back-n* policy, and retransmission resumes from substantially earlier. Similar loss interactions appear as the regular pauses in figure 4. Regardless of the exact period of interaction, the server switching overhead for realistic rates of failure is quite low.

## 5.2 Overhead

Micro-benchmarks of the request fulfillment time for an unloaded server are shown in figure 7. The overhead associated with wedge processing becomes negligible as request size increases to the long-running, large sessions for which our scheme is designed. The impact ranges from an additional 1ms per request (80% overhead) for 1 KByte requests to 12ms (1%) for 8 MByte requests.

The load overhead from the wedge comes from our simple TCP splicing implementation; we do not present its evaluation here. The kernel-assisted techniques mentioned in section 4 or the TCP splicing techniques de-

| Method | Avg. Latency | Avg. Max Clients |
|---|---|---|
| Distance | 34ms | 2146 |
| Optimal | 38ms | 1399 |
| Backoff | 67ms | 1334 |
| Round-Robin | 94ms | 1112 |
| Random | 94ms | 1160 |

**Table 1: Simulation distances and node with most clients for different server selection methods.**

scribed by Cohen, Rangarajan, and Slye [7] can eliminate most of the TCP processing overhead, but not the connection establishment overhead; an in-server implementation can eliminate nearly all of the overhead.

### 5.3 Implosion

To explore the degree to which latency-dependent server selection affected server implosion, we simulated 10,000 clients served by 10 servers. (Other numbers of clients and servers had nearly identical results). The clients and servers were laid out on a random two-dimensional grid to represent the distance between them; possible latencies ranged from 0ms to 250ms.

We first tested two global methods. *Optimal* caps server occupancy at 1400 clients and performs a global minimum-latency assignment to servers. *Round-Robin* guarantees that clients are evenly distributed between (effectively random) servers. Next, we simulated three distributed methods. *Distance* uses only the distance metric to assign servers. *Random* chooses servers purely at random from the failure group. *Backoff* uses an exponential backoff based on the number of clients hosted at the server and the number of outstanding "offers."

The performace of each server selection method is shown in table 5.3. Pure distance assignment has the best latency, but suffers from severe implosion effects. Optimal does nearly as well without the implosion, but has feasibility issues in a distributed environment. Round-Robin and random assignment both result in very even distributions of clients, but have high latency. As expected, our weighted backoff method achieves a nice compromise.

## 6 Related work

Request redirection devices can perform per-connection load balancing, but achieve better performance and flexibility if they route requests based upon the *content* of the request. Many commercially available Web switches provide content-based request redirection [2, 6, 11, 18] by terminating the client's TCP connection at a front-end redirector. This redirector interprets the object request in a manner similar to our wedge, and then passes the request on to the appropriate server. In this architecture, however, the redirector must remain on the path for the duration of the connection, since the connection to the server must be spliced together with the client's connection.

In-line redirection inserts a potential performance bottleneck, and the benefits of TCP splicing [7] and handing off connections directly to end machines within a Web server cluster are well known. Handoff mechanisms were first explored by Hunt, Nahum, and Tracey [13], and later implemented in the LARD (Locality-Aware Request Distribution) system [17]. LARD was recently extended [4] to support request handoff between backend servers for HTTP/1.1 persistent connections [10]. A key component of the LARD implementation is a TCP handoff mechanism, which allows the front-end load balancer to hand the connection off to back-end servers after the load-balancing decision has been made. Similar functionality can now also be found in a commercial product from Resonate [20]. While all three of these mechanisms are transparent to the client, they each require the connection to be actively handed off by the front end to the back-end server. [7]

The need for previous techniques to maintain hard connection state at the front end has made developing Internet telephony systems with reliability equivalent to current circuit-switched technologies quite difficult. A new Reliable Server Pooling (Rserpool) working group has recently been formed by the IETF to examine the needs of such applications. We believe our architecture addresses many of the requirements set forth in the working group charter [16] and described in the Aggregate Server Access Protocol (ASAP) Internet Draft [24].

## 7 Conclusion

We described the design and implementation of a fine-grained failover architecture using transport-layer connection migration and an application-layer soft-state synchronization mechanism. Our architecture is end-to-end and transparent to client applications. It requires deployment of previously-proposed changes to only the transport protocol at the communicating peers, but leaves server applications largely unchanged. Because it does not use a front-end application- or transport-layer switch, it permits the wide-area distribution of each connection's support group.

Experimental results of our prototype Linux implementation show that the performance of our failover system is not severely affected even when connection halts and

[7] Back-end forwarding [4] allows different back ends to serve subsequent requests, but requires the previous server to forward the content (through the front end) back to the client.

resumptions occur every few seconds. Performance decreases only marginally with increasing migration frequency, with an additional contribution dependent on the loss rate of the connection immediately preceding migration. We therefore believe that this approach to failover is an attractive way to build robust systems for delivering long-running Internet streams.

The techniques described in this paper are applicable to a variety of systems, in addition to the traditional end-to-end browser-to-server model. Although our architecture does not require application- or transport-layer switches for routing and health monitoring, it does not preclude them. For example, some commercial products (e.g., Cisco LocalDirector [6]) suggest using two co-located Web switches for redundancy. When used across long-running connections, our solution allows existing connections to be seamlessly migrated between multiple Web switches without adversely affecting performance.

Our source code is available (under GPL) at `http://nms.lcs.mit.edu/software/migrate/`.

## References


[1] Akamai Technologies, Inc. `http://www.akamai.com`.

[2] Alteon Web Systems. Layer 7 Web Switching. `http://www.alteonwebsystems.com/products/whitepapers/layer7switching`.

[3] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *Proc. ACM SIGCOMM '98*, Sept. 1998.

[4] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for P-HTTP in cluster-based web servers. In *Proc. USENIX '99*, June 1999.

[5] Cisco Systems. Cisco Distributed Director. `http://www.cisco.com/warp/public/cc/pd/cxsr/dd/tech/dd_wp.htm`.

[6] Cisco Systems. Failover configuration for LocalDirector. `http://www.cisco.com/warp/public/cc/pd/cxsr/400/tech/locdf_wp.htm`.

[7] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. In *Proc. USITS '99*, Oct. 1999.

[8] Digital Island, Inc. Digital Island, Inc. Home Page. `http://www.digitalisland.net`.

[9] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proc. IEEE Infocom '98*, Mar. 1998.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. HyperText Transfer Protocol—HTTP/1.1. RFC 2068, IETF, Jan. 1997.

[11] Foundry Networks. ServerIron Internet Traffic Management Switches. `http://www.foundrynet.com/PDFs/ServerIron3_00.pdf`.

[12] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Cluster-based scalable network services. In *Proc. ACM SOSP '97*, Oct. 1997.

[13] 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.

[14] D. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proc. IEEE Infocom '98*, Mar. 1998.

[15] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, IETF, Oct. 1996.

[16] L. Ong and M. Stillman. Reliable Server Pooling. Working group charter, IETF, Dec. 2000. `http://www.ietf.org/html.charters/rserpool-charter.html`.

[17] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proc. ASPLOS '98*, Oct. 1998.

[18] Radware. Web Server Director. `http://www.radware.com/archive/pdfs/products/WSD.pdf`.

[19] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *Proc. ACM SIGCOMM '99*, Sept. 1999.

[20] Resonate. Central Dispatch 3.0 - White Paper. `http://www.resonate.com/products/pdf/WP_CD3.0___final.doc.pdf`.

[21] L. Rizzo. Dummynet and forward error correction. In *Proc. Freenix '98*, June 1998.

[22] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. ACM/IEEE Mobicom '00*, pages 155–166, Aug. 2000.

[23] A. C. Snoeren and H. Balakrishnan. TCP Connection Migration. Internet Draft, IETF, Nov. 2000. `draft-snoeren-tcp-migrate-00.txt` (work in progress).

[24] R. R. Stewart and Q. Xie. Aggregate Server Access Protocol (ASAP). Internet Draft, IETF, Nov. 2000. `draft-xie-rserpool-asap-01.txt` (work in progress).

[25] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, IETF, Oct. 2000.

[26] R. R. Stewart, Q. Xie, M. Tuexen, and I. Rytina. SCTP Dynamic Addition of IP addresses. Internet Draft, IETF, Nov. 2000. `draft-stewart-addip-sctp-sigran-01.txt` (work in progress).