Transport Layer Support for Highly-Available Network Services

Florin Sultan, Kiran Srinivasan, Liviu Iftode
Department of Computer Science
Rutgers University, Piscataway, NJ 08854-8019
{*sultan, kiran, iftode*}*@cs.rutgers.edu*

Department of Computer Science Technical Report DCS-TR-429

January 2001

Abstract

We advocate a transport layer protocol for highly-available network services by means of transparent migration of the server endpoint of a live connection between cooperating servers. We propose a client-initiated migration mechanism, integrated in a migration architecture that uniformly addresses various types of events that negatively impact the quality of service to clients. In the resulting architecture, the migration mechanism is independent and decoupled from any migration policies. In our system connection migration can be used to improve performance as perceived by the client, for server load balancing, to survive server failures etc. We examine and address challenges posed by the transfer of connection state needed for migration to both the OS and the application layer.

1 Introduction

1.1 Motivation

The growth of the Internet has led to increased demands on behalf of the users with respect to the quality of services offered over a wide-area network where best-effort service guarantees are the norm. Increased user expectancy also conflicts with increasing load on popular servers that become overloaded, fail, or may fall under DoS attacks. From the clients' perspective all these result in poor quality of service. At the same time, there has been a growing trend in the Internet to view resources as services rather than as servers. The coupling between a resource and its location (IP address) is progressively losing importance. Clients are only interested in obtaining good quality service, no matter from where (geographical location) the service is being provided.

The connection-oriented transport layer protocol of the Internet (TCP), used by a vast majority of services, is oblivious to the quality of the traffic on a connection. The only way TCP reacts to lost or delayed segments is by retransmission to the same remote endpoint of the connection. It does so under the assumption that a missing segment is the result of a problem in the network. As such, the congestion avoidance mechanisms of TCP are concerned only with the well-being of the core network. TCP provides no means to alleviate a degradation in traffic quality (e.g., low throughput, many retransmissions etc.) that may affect the application layer, caused by adverse factors like server overload or network congestion on a given path.

At the same time, TCP creates an implicit association between the server contacted by a client and the service it provides. This is overly constraining for today's Internet service models, where the end user of a service is concerned more with the quality of the service rather than with the exact identity of the server.

These factors motivate us to seek a transport protocol that *(i)* offers a better alternative than the simple retransmission to the same server, which may be suffering from overload, may be down, or may not be easily reachable due to congestion, and *(ii)* decouples a given service from the unique/fixed identity of its provider.

1.2 Our Solution

In this paper we propose a transport protocol that supports *client-initiated connection transfer* among servers providing the same service. We integrate this mechanism in a general migration architecture, with the goal of providing a uniform framework to deal with various types of adverse conditions using the same basic mechanism. In our proposed architecture, connection migration is *initiated* by the *client* side TCP, in response to various *triggers* that can reside either at the client or at the server(s). The same mechanism can be uniformly used whenever migration is deemed necessary for continuing servicing a client or for providing better service to it.

The model of client-server interaction that we envision in our design assumes that there exists a number of hosts (hereby named "servers"), possibly distributed across the Internet, that provide the same *service*. This means that necessary resources may need to be replicated at the servers supporting the service. A client contacts a preferred server using a TCP connection, at the beginning of the interaction with the service. The granularity of the client-server *service session* is the TCP connection established with the preferred server.

At any point during the lifetime of a session, under the control of our scheme, the remote endpoint of the connection may migrate between the cooperating servers, transparent to the client. The servers *cooperate* in order to accommodate a migrating connection by exchanging supporting state. Migration may occur, for example, in response to a decrease in the quality of the traffic seen on the client side of the connection. The only requirement imposed on the client is that we expect it to be willing to renounce control over the identity of its server. In exchange, the service provider is responsible for accommodating the client on a new server in the event that changing the server endpoint becomes necessary. In case of failure to do so, this is a failure to provide the service and the session/connection is aborted.

Our mechanism is not intended to replace TCP or the basic mechanisms TCP uses to recover from losses. We regard it rather as an extension to TCP,

and compatible with it. For example, the client side of our transport protocol could use the number of retransmissions as a metric of the quality of the connection with the current server to trigger migration of the connection to another server. In this case, a proper policy would combine the retransmission-based recovery of the base TCP with the decision to migrate based on this metric.

2 Challenges for the OS

2.1 The Problem

The idea of dynamically and transparently moving the server endpoint of a live connection raises interesting issues for the design of the transfer mechanism and it opens a number of avenues for research. Because the TCP/IP stack is implemented in the OS, migrating a live TCP connection requires transfer of its associated kernel state (e.g., sequence numbers, some unacknowledged segments etc.) from the origin to the destination server. This kernel-level state has the role of reconciling the TCP layer of the destination server with that of the client. At the same time, as Internet services are most often stateful, application-level state describing a *restart point* in the service session must also be transferred to the new server. This ensures that the new server application continues servicing a session from a well-defined, application-specific state.

The problems of TCP in the context of migration stem from the in-kernel data buffering and the active read/write communication model. Because of this, the kernel state of a connection at a given moment in time is most often out of sync with respect to the state reached by the application as a result of reading/writing data on the connection. In addition, the state reached by the server process while servicing a given session cannot be inferred from the in-kernel state of the TCP connection used by that session. However, as the connection is reinstated at the new server after migration, the server process must be able to: *(i)* take over the reincarnated connection at the restart point, and *(ii)* continue to transfer data on it, without altering the exactly-once semantics of TCP data delivery. For example, writes performed at the old server that may be re-issued at the new server due to re-execution starting from the restart poinnt must be correctly identified as duplicates by normal TCP mechanisms and properly discarded. In order to preserve the TCP semantics, one must *synchronize* the server application state associated with a session at the restart point and the in-kernel state of the connection used for data transport by that session.

2.2 Our Approach

Our proposed solution provides a minimal interface to the OS for exporting/importing a *state snapshot* of the server application state. The state snapshot completely describes the point a server has reached in an ongoing session with the client and thus can be used to restart the session after a connection migration. A trivial example of a state snapshot is the value of the current file pointer in a file sent by a HTTP server. The kernel uniquely associates the state snapshot with a pair of sequence numbers in the incoming and outgoing byte streams of the TCP connection. This achieves the desired synchronization of the kernel and application level state.

The most recent state snapshot defines the restart point of the service session for the new server and, along with the in-kernel state of the connection, enables the new server to: *(i)* restart servicing the client session from the state snapshot on, *(ii)* replay reads executed at the old server and for which data can no longer be supplied (retransmitted) by the client-side TCP, and *(iii)* replay writes that were executed by the old server before migration and which are not guaranteed to have reached the client but are reflected in the state snapshot of the old server (so they cannot be re-executed by the new server). Assuming a deterministic process execution model at the server, the above guarantees ensure that the server side of the session can reach the same state it was in before migration.

To replay reads executed at the old server after the most recent state snapshot was taken requires logging them by the kernel. These logged reads are part of the kernel state that must be transferred to the new server. We contend that the amount of this component of the state is small, as in the vast majority of Internet-based services the bulk of the traffic is from server to client.

We believe that, in general, the amount of application-level state to be transferred (the size of the state snapshot) should not be a concern. Because the moment when the snapshot is taken is under the control of the server application, it can be chosen such that to reduce the size of the snapshot for a connection. While this may not be possible for all applications, it may also make our method of state transfer feasible for a large number of applications.

We may extend our scheme by including in the application-level state of a connection, beside local server resources, the connections established by the server for servicing the client. Then, when migrating the server endpoint, the connections opened on behalf of the client must be also recursively migrated.

2.3 Discussion

Our approach presents challenges to the server application. It must obey a certain programming discipline by calling primitives for exporting/importing to/from the OS the application-level state associated with a potentially migrating connection. Also, the application may have to exploit certain tradeoffs. One possible tradeoff is between the size of the snapshot and the moment it is taken: delaying a snapshot may increase the amount of read data to be logged.

Although we require changes to server applications, we believe that in general the programming effort involved should be fairly low, and expect the mentioned demands not to be very intrusive for the application logic. Moreover, adhering to a certain programming discipline and API can be viewed as the effort a server writer may want to invest in order to take advantage of dynamic server-side migration of live TCP connections.

The connection migration mechanism we propose can be used *(i)* on the client side, to dynamically switch to another server when the current server does not provide a level of satisfactory service, or *(ii)* on the server side, for example to implement a load balancing scheme by shedding load from existing connections to other less loaded servers, or an internal policy on how the content should be distributed among groups of clients.

In addition, we believe that our scheme can be used to alleviate the impact on existing connections of certain types of DoS attacks (SYN flooding, the process table attack) by shifting connections from the host under attack to alternate servers. While the migration mechanism may need to use resources on the attacked machine (which, depending on the severity of the attack, may or may not be available), connection migration provides in any case a better alternative than the loss of existing connections.

The features of our approach that distinguish it from other approaches are:

- It is application independent. Compared to schemes like [7, 8] it is general and flexible, in that it does not rely on knowledge about a given server application or application-level client-server protocol.
- It is light-weight. Compared to TCP fault tolerance schemes like [2], which work only for a whole process context, our mechanism operates at the granularity of individual connections. It thus can be used in a TCP fault tolerance scheme where individual connections can be recovered separately after a failure, possibly on different servers.
- It is symmetric with respect to and decoupled from any migration policy. Compared to approaches like [7], it enables both the client and the server to trigger the migration, and it enables a server to control the migration process.

3 Migration Model and Architecture

In this section we present the basic model underlying our connection migration mechanism. We are not concerned here with policy or implementation-related issues like how to decide when to migrate, what events can trigger a migration, how/when is the connection state transferred from the origin to destination server etc.

In our model, connection migration involves only one endpoint of the connection (the server side), while the other endpoint (the client) is fixed. Migration may occur multiple times throughout the lifetime of a connection. The mechanism for connection migration reincarnates the migrating endpoint of the TCP connection at the destination server and establishes a restart point for the server application, by using kernel and user level state associated with the connection at the origin server. In describing our model, it is not essential how the connection state is transferred to the destination server: it may be eagerly propagated by the originating server (for example at regular intervals), it may be transferred on demand (at the request of the destination server, when the migration is initiated by the client), or it may be even stored with the client and provided by it when initiating a migration.

Our goal is to develop a general migration architecture and, within it, provide a mechanism that can be uniformly used to react to various events that may require a connection to migrate, regardless of *where* the decision to migrate is taken. For example, a load balancing decision is always taken at the server side, while migration as a result of perceived bad performance can be decided on the client side.

3.1 Migration: Triggers and Initiator

Figure 1 shows the three entities involved in a migration: the origin server S_1, the destination server S_2, and the client C. Recall that one of our goals is not to change client applications, therefore the levels at which the three entities operate are *not* symmetric: the entity labeled C actually represents actions supporting the transparent migration of the remote endpoint that take place in the TCP/IP stack on the *client process*'s side of a connection. However, actions of S_1 and S_2 may originate at either the transport or application layer.

The basic idea in our architecture is to distinguish between the *trigger* and the *initiator* of a migration.

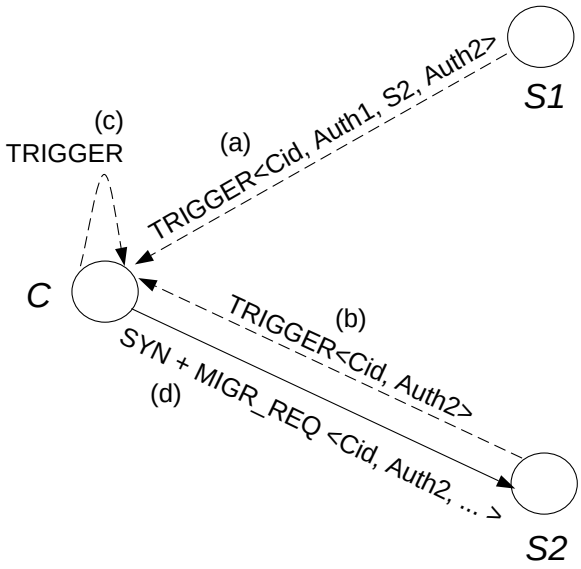

Figure 1: *Connection migration architecture based on client-initiated migration. The connection Cid initially established between client C and server S_1 migrates from S_1 to S_2. Migration triggers can be either a server (cases a, b) or the client itself (c). The client initiates the migration by attempting a new connection to S_2 with the SYN packet (d).*

The trigger is the entity which decides that migration of the remote endpoint of a client connection is required. The initiator is the entity that actively starts the process of reinstating the connection at the destination server.

In our architecture, the *trigger* of the migration can be any of the three parties involved. However, in all cases, the sole *initiator* of the migration is always the client C. The role of the trigger is just to inform the client that it must migrate a certain connection in which it is involved, while the role of the client is to actively take the steps to do so. The advantage of having only the client initiate the migration lies in using a uniform mechanism, regardless of which of the parties triggers migration.

When an event that prompts a connection migration occurs, the trigger (if remote) sends a special TCP segment to C, with an option $TRIGGER =< Cid, Auth_{trig}, [S_{dest}, Auth_{dest}] >$ to signal a request for migration (case (a) or (b) in Figure 1). The option carries the 4-tuple connection identifier Cid (IP addresses and ports for both C and S_1 endpoints), along with authentication information $Auth_{trig}$ that identifies the trigger as legitimate for asking the client to initiate a migration. In addition, it may contain the destination server's IP address and an authenticator for the migrating connection at the new server (for example, if the trigger is S_1, Figure 1 (a)). The segment carrying the $MIGRATE\_TRIGGER$ option can be sent in-band on the existing connection if the trigger is the current server. If the trigger is another host, the option is sent with a special SYN segment on the port used by Cid (the option itself discriminates the SYN for special processing).

3.2 Servers as Triggers

Using Figure 1 as example, suppose that a load balancing scheme is implemented by dynamically migrating connections between servers. Once a load balancing decision is made at a server, it just becomes a migration trigger for a connection, and sends a $MIGRATE\_TRIGGER$ segment to the client. Then the client initiates the migration by connecting to the destination server as seen in Figure 1, *(d)*.

Two styles of load balancing interactions are possible, depending on where the load balancing decision is made. With reference to Figure 1, *(a)* S_1 could decide to push some connection away to S_2, or *(b)* S_2 could decide to pull a connection from S_1. In the latter case S_2 must know, through some external mechanism, the Cid of the connection it wants to pull from S_1. In both cases the trigger's decision is independent of and transparent with respect to the client-initiated migration mechanism. Moreover, the scheme can be extended to allow another host, different from the origin and destination servers to act as migration trigger.

As another example, the mechanism can be used for migrating the connection in order to recover from a failure of S_1. In this case, S_2 can become a migration trigger if responsible with monitoring and backing-up S_1, in case it detects its failure.

3.3 Client as Trigger

The client C can itself act as a (local) migration trigger as shown by case *(c)* in Figure 1. This case requires the cooperation of the original server, as the client alone has no way of knowing what the new server(s) should be. To accomplish this, during the initial handshake of the connection setup, S_1 sends a special option $MIGRATE\_SUPPORT =< S_{list}, Auth_{dest} >$ where S_{list} is a list of IP addresses of possible migration destinations, and $Auth_{dest}$ is an authenticator for the migrating connection at the intended destination. This enables C to act as a trigger and provides it with destination address(es) and the authenticator needed to initiate the migration.

The client's decision to trigger a migration may be based on its perceived performance of the service received from the current server (S_1). The quality of the service can be estimated according to some service-specific metric, e.g., associated with the well-known

server port. For example, a file transfer service can use the perceived incoming throughput as performance metric. A degradation in performance could be used as the *local* event to trigger migration. For example, a loss in throughput or repeated retransmission timeouts may be a sign of a problem in the network or of the failure of server S_1.

3.4 Initiator

Once it has been informed (or it has decided) that a migration of Cid is needed, the client initiates the migration by attempting a new connection with the destination server, as shown in Figure 1, *(d)*. A special option $MIGRATE\_REQUEST =< Cid, Auth_{dest} >$ is sent by C with the SYN, to request migration of the connection Cid to the target server. After proper authentication, the state associated with the connection is reinstated at the destination and a SYN/ACK is returned to the client. The SYN/ACK signals to the client that the destination has accepted the migration. At this point, the client TCP completes the migration by transparently switching the client process to the local endpoint of the new TCP connection and continues data transfer on it.

4 Related Work

TCP/IP connection hand-off protocols have been used either for mobility extensions to TCP/IP [4, 5, 6] or for request distribution in clusters [3].

The mobile TCP approaches do not consider the task of migrating the connection endpoints between physically distinct machines. They either rely on directly using the full connection state maintained at the physical endpoints [4, 5], or on restarting a previously established connection after a change of the IP address by reusing the state of the old connection after proper authentication [6].

[3] proposes TCP connection hand-off in clustered HTTP servers for distributing incoming requests from a front-end machine to the server back-end nodes. It is limited to a single hand-off scheme, where a connection endpoint can migrate only during the connection setup phase. Multiple hand-offs of persistent HTTP/1.1 connections are only mentioned as an alternative, but no design or implementation is described. Even in the multiple hand-off scheme, the granularity of migration of live connections is application-dependent: a connection can only migrate after fully servicing a HTTP request.

[2] describes a scheme for transparent (masked) recovery of a crashed process with open TCP connections. A wrapper around the TCP layer intercepts and logs reads by the process for replay during recovery, and shields the remote endpoint from the failure. Their system focuses on masking a fault, while we address transparent, fine-grained and light-weight migration of a connection endpoint. Although their scheme could be used for connection migration, this would require coarse-grain migration of the full process context. Our system allows migration only of a connection endpoint, while the process can continue to run.

[8] discusses a technique for fault-resilience in a cluster-based web server using a front-end distributor to monitor the state of client connections serviced by back-end nodes. In case of node failure, the distributor restores the serviced connections on another node. The technique is application specific and makes the distributor a single point of failure and a potential bottleneck. While their scheme could be used for connection migration, this is only possible inside the cluster. Our migration scheme does not have these drawbacks and works over wide area.

The Akamai Technologies, Inc. [1] provides a commercial solution for highly-available content distribution over Internet, using HTTP and DNS to redirect clients' requests to proprietary content servers interconnected by a private network. While addressing the same problem, our scheme is different: it does not depend on a support infrastructure and it is flexible, allowing a service provider to make its content available to clients according to its own policies.

[7] presents a work that is closest to ours. It describes a failover architecture that enables migrating a HTTP connection endpoint within a pool of support servers that replicate per-connection soft transport and application state. Endpoint migration is initiated only by the server, on events like server failure or overload. Their architecture adds an HTTP aware module at the transport layer that extracts information from the application data stream to be used for connection resumption. While this design leads to no change in the server application, it is not general, as the transport layer must have knowledge of upper layer protocols. The design depends on the information in the application data stream being: *(i)* sufficient for describing application state associated with a connection, *(ii)* compatible or up-to-date with respect to the in-kernel module, and *(iii)* always accessible, i.e., application traffic can only be (unencrypted) clear text. In our scheme, we expect the server application to change, assuming a transport protocol that allows for dynamic connection migration. We allow the server to specify to the kernel the state needed to resume the data transfer in case of connection migration. This application specific state is completely opaque to the kernel and the transport protocol.

5 Status

We have designed the client-initiated migration mechanism and we are implementing it on a FreeBSD platform. We have also designed the server API and are in the process of implementing it.

References

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

[2] L. Alvisi, T.C. Bressoud, A. El-Khashab, K. Marzullo, D. Zagorodnov. Wrapping Server-Side TCP to Mask Connection Failures. *Technical Report, Department of Computer Sciences*, The University of Texas at Austin, July 2000.

[3] M. Aron, P. Druschel, W. Zwaenepoel. Efficient Support for P-HTTP in Cluster-Based Web Servers. *USENIX '99.*

[4] Ajay Bakre, B. R. Badrinath. Handoff and System Support for Indirect TCP/IP. *Second Usenix Symposium on Mobile and Location-dependent Computing*, April 1995.

[5] H. Balakrishnan, S. Seshan, E. Amir, R. H. Katz. Improving TCP/IP Performance over Wireless Networks. *1st ACM Conference on Mobile Computing and Networking*, November 1995.

[6] A. C. Snoeren, H. Balakrishnan. An End-to-End Approach to Host Mobility. *6th ACM MOBICOM*, August 2000.

[7] A. C. Snoeren, D. G. Andersen, H. Balakrishnan Fine-Grained Failover Using Connection Migration. *Technical Report MIT-LCS-TR-812*, MIT, September 2000.

[8] C. Yang, M. Luo. Realizing Fault Resilience in Web-Server Cluster. *SuperComputing and Networking 2000*, November 2000.