Unveiling the Transport

Jeffrey Mogul, Lawrence Brakmo, David E. Lowell, Dinesh Subhraveti, Justin Moore
HP Labs, Palo Alto
{Jeff.Mogul,Lawrence.Brakmo,David.Lowell,Dinesh.Subhraveti,Justin.Moore}@hp.com

Abstract

Traditional application programming interfaces for transport protocols make a virtue of hiding most internal per-connection state. We argue that this information-hiding precludes many potentially useful application features and performance optimizations. We advocate a disciplined, portable, and secure interface that gives applications both "get" and "set" access to transport connection state.

1 Introduction

Most modern operating systems provide a protocol-independent application programming interface (API) to transport protocols such as TCP or SCTP. This API, often based on the BSD *socket* abstraction, makes a virtue of hiding most internal per-connection state. Applications use abstract "connections" with high-level characteristics (e.g., "reliable stream") guaranteed by the protocol implementation, and remain ignorant of internal details such as sequence numbers, round-trip time estimates, and transport-level options.

The information-hiding inherent in a socket-like transport API promotes application portability between transport protocols, and between different implementations of the same transport. If the application cannot see the internal state for a connection, it won' t have any dependencies on this state. Other operating systems, such as Plan 9 [18], go even further than BSD in making a virtue of hiding information.

This approach has reached its limits. It worked fine for simple applications, such as Telnet, FTP, and email. But traditional hidden-state transport APIs preclude many potentially useful application features and performance optimizations. In this paper, we describe a number of application techniques that cannot be implemented using hidden-state APIs, but that could exploit exposed transport state.

We advocate that a traditional API, such as sockets, could be extended with a disciplined, portable, and secure interface to give applications both "get" and "set" access to transport connection state. The provision of "set" access is a radical suggestion, and we address both its justification and its safety.

In this paper, we focus on TCP, rather than on transports in general. Other transports are either uninteresting (e.g., UDP has essentially no "per-connection state") or are too novel (e.g., SCTP implementations are available [20] but we aren' t sufficiently knowledgeable about the protocol or the implementations). The principles we discuss with respect to TCP should be applicable to other transports, and the API proposed in Section 5 is mostly transport-generic.

2 A few scenarios

We start by presenting a few scenarios to motivate the rest of the paper. Later, in Section 4, we will discuss a larger variety of potential applications.

2.1 Process migration

Suppose one wants to migrate a running process from one host to another, without requiring explicit migration support in the operating system, and transparently to all communicating peer processes on other hosts. Milojicic *et al.* argue that user-level implementations of migration have some benefits over kernel implementations, but point out that moving connection endpoints can make this difficult [14]. For example, when transferring a live TCP connection, the TCP sequence numbers must be preserved. This is currently extremely difficult for a user-level implementation of process migration [1].

In our approach, the application (or a user-level library acting on its behalf) on the migration source host would use the "get" feature of our proposed API extension to extract all the necessary TCP connection state for the process. (See Section 3 for a discussion of what state this entails.) This state would then be shipped to a new process at the migration target host, which would create new sockets and initialize them with the imported TCP connection state, via the "set" feature of our API. Of course, many other challenges face such a migration mechanism; we believe (from implementation experience [16]) that they are all surmountable, but we lack space to discuss them here.

2.2 Adapting Web content to the network path

A widely-used Web server on the Internet might serve clients with dramatically varying network connectivity. Some people use dialup networks; others use cable modems. The available bandwidths span several orders of magnitude, posing a problem for a site designer who wants to provide rich detail for well-connected users while not subjecting dialup users to enormous delays.

If the Web server could make a crude estimate of the network path as soon as the client establishes its connection, the server could then deliver content whose complexity is adapted to the path bandwidth – without requiring the user to click to choose between "high bandwidth" and "low bandwidth" versions of the site [15].

A crude estimate that simply clusters clients into "fast" and "slow" categories should allow a server to choose between a pair of page versions [11]. Such an estimate could derive from the initial round trip time (RTT) measurement for the connection, which is available as soon as the client's request arrives. A single RTT measurement might be an inaccurate predictor of the path bandwidth, but since a TCP connection's initial throughput is usually RTT-limited, not bandwidth-limited [4]), it could be sufficient.

Making TCP connection state information, such as the RTT estimate, the congestion window, the retransmission rate, and other indirect estimators of the path quality, available to an Internet server could allow it to adapt its behavior automatically.

3 Categories of transport state

We find it useful to categorize transport state according to several axes:

Hard state vs. soft state: The *hard* state of a transport connection must be preserved (i.e., across a checkpoint or migration) in order for the connection to continue transparently to the remote peer. Hard state includes the TCP port number, state-machine state (e.g., ESTABLISHED or TIME_WAIT), sequence and acknowledgement (ACK) numbers, and some TCP options such as Window Scale and MSS. *Soft* state could be discarded without affecting correctness, and can be recovered or rediscovered, although its loss might reduce performance. Soft state includes the estimated RTT, current settings of connection timers, and the congestion window.

Fixed, varying, or connection-initialization state: Certain connection state is inherently *fixed* from the initial creation of a connection. For TCP, this includes the port numbers and IP addresses. Other state, such as sequence numbers, varies during the entire lifetime of the connection. A third category of state, such as the TCP MSS, is allowed to vary only during the initialization of a connection. We separate the *fixed* and *connection-initialization* categories because the former can never be changed, but the latter might change (under the control of the remote peer) during initialization, which might affect how the application interacts with this state.

In addition to these state values, we also need to give the application control over whether they can change asynchronously. That is, when the application is getting or setting connection state, in many cases it needs atomic access to the entire state, and it might need to avoid asynchronous state changes that might result from packet arrivals or timer expirations. Thus, we define a sort of meta-state for the connection that can be one of *in-progress*, *quiescing*, or *frozen*. The application must be able to request that the protocol stack quiesce an in-progress connection, to discover when quiescence is complete (i.e., that the connection is frozen), and to resume from the frozen state.

4 Applications of this approach

In this section, we sketch a variety of applications that could benefit from more access to transport connection state. We have divided the list of applications into several major categories, but this categorization is neither exhaustive nor definitive.

Our goal in presenting this set of applications is to motivate the provision of a simple and generic facility for user-level access to transport connection state. Many of these applications could instead be enabled using problem-specific operating system extensions, but we advocate the adoption of a generic mechanism because it should be useful even for applications that we haven' t thought of.

4.1 Connection persistence over failures and migrations

Traditional APIs bind transport connections to specific processes. The process abstraction, while useful, sometimes overdetermines the binding of connections and other resources. It can be valuable to recover the state of a process (i.e., using a checkpoint) for fault tolerance [9, 12], or to move the state of a process (i.e., process migration [14]) for either fault tolerance or load balancing.

Both checkpointing and process migration require specific techniques to preserve extant transport connections. We focus on techniques implemented in user-level code.

4.1.1 Process migration via a user-level library

Process migration can be implemented as a user-level library with relatively little explicit kernel support, but migrating active transport connections can be difficult. We introduced this application in Section 2.1.

Note that in addition to access to hard connection state, such as TCP sequence numbers, a user-level library that migrates connections would also need a way to freeze the transmission of TCP acknowledgments. An ACK allows the remote peer to discard its buffered data; if our host sends an ACK *after* we snapshot its connection state, then we could lose data arriving after that snapshot. (It might not be necessary to freeze the processing of arriving ACKs, since losing this information causes extraneous retransmits rather than data loss.)

Wholesale migration of processes from one host to another also implies the need for a means to move the host's IP address. (If both ends of the connection are controlled by the process migration system, this might not be necessary.) There are several possible solutions to that problem, such as broadcasting an Address Resolution Protocol message to change the *(IP, Ethernet)* bindings. Alternatively, one could perhaps support changing the IP address, through the *Migrate* TCP options proposed by Snoeren and Balakrishnan [22].

All of the considerations above, except for IP address migration, also apply to a process checkpointing facility, for use in preserving processes across failures such as system crashes.

4.1.2 Reading unavailable input data

If a process with open transport connections is being migrated or checkpointed by user-level code, the checkpoint must capture received data that the kernel has already ACKed but not yet delivered to the application. (Either the application has not yet made a system call to read the data, or the call has not fully completed.) For transport implementations using only cumulative acknowledgments, such as the original TCP standard, this is relatively simple: the library code freezes ACK transmission and then uses the standard *read()* system call to read all of the data up to the cumulative ACK limit.

Modern TCP implementations, however, support selective acknowledgment (SACK) [13], which allows the receiver to acknowledge data even if there are holes in the arriving TCP stream. The semantics of the *read()* system call prevent it from reading past such a hole, making it impossible for a user-mode library to capture any selectively-acknowledged data following the first hole.

This is not a correctness problem for SACK, which is specified to prevent the sender from dropping data before the corresponding cumulative ACK, although it might be a problem for a future transport that lacks this requirement. It might also be a performance problem when migrating or checkpointing an application, such as a server with lots of open data-receiving connections.

Thus, a user-level migration/checkpoint facility might benefit from an API allowing it to capture and restore acknowledged but unavailable input data.

4.1.3 Making TCP sends redoable

Typical rollback-recovery systems require that all application events are either natively “undoable” or “redoable,” or can be made so by the recovery system [12]. Most application events do satisfy this requirement. For example, modifying data in memory or writing to a non-shared file are both intrinsically redoable events, and both can be made undoable through appropriate undo-logging in the recovery system.

However, because of a message's effect on other processes, a TCP send cannot (easily) be made undoable. Nor can TCP sends be made redoable at user level. If a process sends a TCP message, fails, recovers to an earlier execution point, then attempts to resend the same message, TCP will transmit the “resent” message with a new range of sequence numbers. The message recipient will see this as a new message rather than a retransmission.

The problem for user-level redo is that the send sequence number is not exposed by the traditional TCP API. Existing recovery systems work around this problem by intercepting and inspecting outgoing TCP frames [1], using a special TCP implementation at the client and server [21], rolling back the state of the message receiver [23], or spoofing the application into using a user-level, TCP-like transport over UDP [12].

An API that exposes the sequence number would permit a user-level recovery system to checkpoint it along with the rest of the process state. An API that also allows “set” access to the sequence number would permit the recovery code to restore this to its last committed value. Both “get” and “set” are necessary to render TCP sends redoable, and to enable a user-level recovery system to correctly recover applications with TCP connections.

4.1.4 Delaying ACKs until receiver commits data

Many failure recovery systems are based on message logging [9], recording the non-deterministic events executed by a process, such as message receptions. The recovery system can replay the logged events to the recovering application, causing it to deterministically recompute its pre-failure state.

A user-level implementation of a pessimistic logging protocol intercepts each received message and writes it to stable storage before delivering it to the application. Since the TCP sender deletes its buffered copy of the message upon receiving an ACK, the logger must prevent this ACK until the message has been safely logged [1]. Otherwise, a failure after the ACK and before the log write would lead to inconsistency after recovery[1].

If the TCP API allowed user-level code to decide whether to send an ACK for a particular byte in the TCP sequence space, this would permit a user-level logging system to delay ACKs until messages are safely logged. Note that delaying an ACK beyond limit of the smaller of the receiver and sender windows could lead to deadlock, unless the ACK-delaying mechanism includes a timeout.

4.1.5 Hiding a recovering system

If a remote peer might time out while a failed system is recovering, it might be useful to create the illusion that the TCP connection is still alive. One way to do this, similar to a technique used by Alvisi *et al.* [1], is to employ a "helper system" that fakes activity on one or more connections while the real system is recovering.

A helper system that knows the current sequence and acknowledgment numbers for the crashed end of a TCP connection can send periodic keep-alive packets to the remote peer, which appear to come from the crashed host. It probably does not need fully up-to-date sequence number values, since use of somewhat old values, while appearing to be delayed duplicates, should still convince the remote peer of the liveness of the system. So, the helper system can obtain sequence and acknowledgment numbers lazily and asynchronously from the protected system, while it is not crashed. (Alvisi *et al.* also suggest simulating a closed receive window, to prevent the wasteful transmission of data that will be discarded.)

4.2 Performance adaptation

Most connection state needed to support persistence is hard state. Soft connection state, in contrast, is most useful in improving application performance, and especially in adapting this performance to network conditions.

4.2.1 Adapting content to path bandwidth

Section 2.2 introduced the potential for adapting Web content to the capacity of the network path to the client, and using subsets of the "soft" TCP state to quickly (if crudely) estimate that capacity.

This technique, if based on timing information such as the RTT, may require the use of finer-grained timers than typically employed in TCP stacks. Since most dialup modems impose one-way delays of 50 msec, independent of distance [6], an RTT well below 100 msec clearly indicates a non-dialup (i.e., potentially high-bandwidth) TCP connection. RTT measurement in 4.4BSD-based systems uses a 500-msec clock tick, far too coarse for discriminating between dialup and other paths. Brakmo and Peterson described an efficient way to compute RTTs at much better precision [3]. One might also estimate bandwidth using a "packet-pair" approach [10], if the API can provide fine-grained arrival timestamps for the most recent packets (and their lengths).

4.2.2 Checking output buffer size

Some applications (e.g., streaming media servers) can adapt to changing network congestion by dynamically increasing or decreasing the compression of transmitted content. There is no value to data that is stuck in the sending host's output buffer for lengthy periods; the client will discard it, as too stale, when it arrives. A server that can detect congestion could reduce its output rate consistent with network capacity, thus delivering timely (if somewhat degraded) content to the client.

A TCP-based streaming server can easily detect congestion on a specific path by monitoring the amount of buffered output data. If the buffer size is growing (over a suitable measurement interval) then data is being sent faster than the network can carry it. Measuring buffer consumption by blocking on a *send()* system call is inefficient and inaccurate; an explicit API feature that returns the number of buffered, unacknowledged bytes would be quite useful.

A pending-output-bytes measurement mechanism might also be useful for an application (such as a Web server) wishing to avoid overcommitting kernel buffer space, or to preserve it for high-QoS customers.

4.3 Implementation techniques

The final category of applications involves novel implementation techniques.

4.3.1 Moving connections between TCP stacks

Traditionally, each host has just one implementation of the TCP stack. However, in certain circumstances a host might need to move a live connection between one of several implementations.

For example, if the TCP implementation is being updated to fix bugs or add features, on a high-availability system it might be desirable to run the new implementation temporarily in parallel with the old one (with some coordination!), shift live connections to the new implementation, and then disable the old one.

Or if the network interface provides "TCP offload", it might be desirable to shift live connections between the offloaded and software implementations (e.g., if resource limits of the offloaded implementation allow it to host only a small subset of the connections.)

[1] Optimistic logging [23] avoids this risk, at the expense of a complicated protocol for rolling back the sender after a failure.

Moving connections for these reasons is similar to moving them for process migration or checkpointing, and has similar requirements. Stack-shifting, however, places a premium on the portability of connection state; that is, it requires an external form that does not reflect the details of a specific implementation.

It is not clear if stack-shifting is best done inside or outside the kernel, but a user-level solution is attractive because (if done via a uniform API) it works seamlessly across any pair of stack implementations.

We are not aware of any implementations that shift live connections between stacks on the same host. However, *Migratory TCP* does implement connection-shifting between stacks on different hosts [24], which should be of similar difficulty.

4.3.2 Absolute sequence numbers in MPA

"Marker PDU Aligned Framing for TCP" (MPA) is a proposed framing layer, between TCP and an upper layer that supports direct (non-buffered) placement of received data into memory [7]. MPA uses *markers* to allow hardware-based direct placement even when TCP segments are received out of order. The details of MPA are beyond the scope of this paper, but the marker mechanism is of interest.

MPA markers are inserted every 512 bytes in the TCP sequence number space. This deterministic spacing allows the TCP receiver to locate the markers in any packet. The specification calls for inserting markers at 512-byte boundaries *relative* to the start of the connection. Eiriksson [8] pointed out that this requires a hardware implementation of MPA to obtain some protocol control block information for each arriving packet before it can locate the markers; such lookups are costly. He proposed instead that markers should appear at 512-byte intervals in the *absolute* sequence number space (i.e., when the sequence number mod 512 is zero).

While this proposed modification had both costs and benefits, the main argument against it was that standards-body considerations require that MPA be implementable in user-level code, over an existing TCP stack, and existing stacks do not expose TCP sequence numbers. If widespread APIs *had* exposed TCP sequence numbers, then MPA could have used absolute sequence numbers.

This scenario does not imply that adopting our proposals for the TCP API would change the design decision for MPA; it is far too late for that. We include this example only to show how in an alternate universe, where all TCP APIs already provide access to the sequence number, this design for MPA would be feasible and perhaps would have been adopted.

5 A disciplined API

In this section, we propose an exposed-state API for TCP. A similar API should be feasible for other transports. For reasons of space, we omit many details (some of which are in any case not yet clear to us).

We call this a "disciplined" API because, while it exposes state previously considered best left hidden, we have tried to expose only state, not implementation details. Ideally, our API should permit portability of user-level code, and perhaps even migration of active connections, between different operating systems.

The API should also allow evolution of transport protocols and kernel implementations. That is, an application using this API that works before the introduction of a new protocol feature should continue to work after its introduction, and should not defeat the use of that feature.

We note that although our API proposal does require changes to the kernel's transport protocol implementation, these changes are relatively simple. Moreover, by enabling user-level implementations of many new functions, this one new API supplants many other potentially useful, but possibly complex, kernel modifications.

5.1 Connection identification

The traditional BSD operations (such as *getsockopt()*) identify connections using a file descriptor. This limits those operations to the process that is using the connection (or its children). We propose, for our state-access operations, identifying connections instead by the protocol-ID (such as TCP, SCTP, etc.) and the corresponding address tuple – *(src-addr, src-port, dst-addr, dst-port)* for TCP. This gives any process with the same user-ID as the connection owner, or a root-privileged process, "get" and "set" access to connection state. This in turn enables the use of "helper processes" to assist with functions such as checkpointing or migration. It also allows cleaning up the frozen connections of a dead process.

For compactness, we use the term *conn-ID* below to refer to the *(protocol, src-addr, src-port, dst-addr, dst-port)* tuple.

There is one exception to the conn-ID approach: it cannot be used to bind an existing address tuple (e.g., for a migrating connection) to a new socket, since that new socket has no tuple. The API will need a special call, using a file descriptor, for this.

5.2 Access to state values

We want to avoid tying our API to a specific stack implementation, or to a specific point in the lifetime of a stack. Since different stacks will naturally provide different support for certain soft state, and certain optional components of hard state, the API should not use an internal representation of transport state. Instead, we advocate a *(keyword, datatype, value, flags)* representation, where "keywords" are an enumeration type, and "datatype" is analogous to the simple types of the C language, including arrays. The "flags" mark a returned state item as hard vs. soft, initialized vs. uninitialized, etc.

The API would include one state-reading operation *transport_state_get(conn-ID, tuple-count, tuple-vector)*. The tuple-vector (with tuple-count entries) both indicates what state items are wanted, and returns their values (if available). Because one call returns multiple items, this should reduce the overhead of using the API. The *transport_state_get* operation should also provide a means (such as wild-carding) for the application to get all hard connection state, even for keywords unknown to the application; this supports protocol evolution.

Similarly, the operation *transport_state_set(conn-ID, tuple-count, tuple-vector)* updates the connection state from the values in the tuple-vector.

5.3 Connection progress

The *transport_freeze(conn-ID)* operation sets an internal per-connection flag preventing the transport stack from taking any action on the connection. Timeouts are deferred (not lost); arriving packets might be either buffered or dropped. The *transport_resume(conn-ID)* operation clears this flag, releases any deferred timers, and starts the processing of buffered packets.

5.4 Buffer manipulations

The *transport_read_pending(conn-ID, buffer, bufsize, bufvec-array, count)* operation returns, into the buffer, all received data that has been ACKed, even if there are holes in the sequence space. Bufvec-array is an array of *(offset, pointer, length)* tuples representing the extents of received data. The corresponding *transport_restore_pending(conn-ID, buffer, bufsize, bufvec-array, count)* operation puts data back into the "unread" portion of the connection's input buffer. The *transport_buffer_purge(conn-ID)* operation deletes all buffered input and output data for the connection.

5.5 Timing information

Section 4.2.1 speculated that packet-pair timing information could help with server adaptation. A transport stack could efficiently support this by keeping a small per-connection ring buffer of recent packet arrival timestamps and lengths. Our API could allow the server application to read this buffer.

5.6 Security

In general, we have not yet found any obvious security holes created if a process (or its designated helpers) can modify the internal state of its own connections, via the API we have described. The one exception is the potential for performance-related mischief, such as denial-of-service attacks. Savage *et al.* have described how a misbehaving TCP receiver can violate congestion control norms [19] and our API would make that easy.

We have considered several solutions to this problem (beyond the protections in [19]). It might be possible for the kernel to cryptographically sign a subset of the state it exports, and then to refuse to import an improperly signed state vector. However, this approach might be too rigid, and relies on secure key management (across multiple hosts, for a migration system).

Alternatively, the API could restrict the setting of certain state items to super-user processes. User-level implementations that move or update state would require the help of such a "chaperone" process, via the conn-ID approach of Section 5.1, rather than allowing a regular process to directly update its connection state.

We suspect the security analysis of our approach will require more work, especially with respect to transport protocols other than TCP.

6 Related ideas

Several previous projects have addressed the appropriate level of application-level exposure of operating system internal state, and the associated policy/mechanism separation issues.

The V++ *Cache Kernel* [5] maintained kernel caches of various kernel state (for example, page table entries) but used handlers outside the kernel to implement almost all decisions and state manipulation. This design seems well-suited to checkpointing and process migration. Since V++ was a micro-kernel, it had no kernel network stack and so the Cache Kernel mechanism did not apply to transport connections.

In the InfoKernel approach [2], a traditional (monolithic) kernel is modified to export abstract information about internal state, in order to allow user-level code to influence kernel policies. For TCP connections, InfoKernel provides both sequence-number values and some per-packet timing information (similar but not identical to our proposal in Section 5.5). However, the InfoKernel philosophy does not include an explicit state-setting API; while InfoKernel does allow user-level code to emulate TCP Vegas [3] behavior on top of a TCP Reno kernel, it cannot support user-level process migration or checkpointing.

As noted in Section 5.6, an application that can update transport connection state could generate network-unfriendly packet flows [19]. Patel *et al.* have shown, in their work on remotely upgrading transport protocol implementations [17], that a kernel can enforce "TCPfriendly” flow rate restrictions on untrusted protocol code. Use of this kind of approach in conjunction with our API might suffice to protect the network against excessive traffic.

7 Summary

We have argued that many interesting applications could use an API that exposes per-connection transport state. We have attempted to explain the kinds of state, and kinds of manipulations, that would be necessary or useful. We sketched a simple and portable API extension that should meet these requirements.

Many of the scenarios we have described for exploiting our approach could be resolved by application changes rather than by exposing connection state. We assert that while changing the applications might be the “right” solution, if one wants to provide application-generic services such as recovery or migration, the applications must be accepted as they are.

References

[1] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side TCP to mask connection failures. In *Proc. INFOCOM*, pages 329–337, Anchorage, AK, April 2001.

[2] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with infokernel. In *Proc. SOSP*, pages 90–105, Bolton Landing, NY, Oct. 2003.

[3] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[4] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *Proc. INFOCOM*, pages 1742–1751, Tel Aviv, Israel, March 2000.

[5] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proc. OSDI*, pages 179–193, Monterey, CA, Nov. 1994.

[6] S. Cheshire. Latency survey results (for “It's the Latency, Stupid”). `http://www.stuartcheshire.org/rants/LatencyResults.html`, 1996.

[7] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier. Marker PDU aligned framing for TCP specification. Internet-Draft draft-culley-iwarp-mpa-03, IETF, June 2003. This a work in progress.

[8] A. Eiriksson. Relative location of MPA markers considered bad for pipelining. Personal comm., June 2003.

[9] E. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[10] S. Keshav. The packet pair flow control protocol. ICSI Tech. Rep. TR-91-028, Intl. Computer Science Institute, Berkeley, May 1991.

[11] B. Krishnamurthy and C. E. Wills. Improving Web performance by client characterization driven server adaptation. In *Proc. WWW-11*, Honolulu, HI, May 2002.

[12] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proc. OSDI*, pages 289–303, San Diego, CA, Oct. 2000.

[13] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. RFC 2018, IETF, October 1996.

[14] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[15] J. Mogul and L. Brakmo. Method for dynamically adjusting multimedia content of a Web page by a server in accordance to network path characteristics between client and server. US Patent 6,243,761, June 2001.

[16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proc. OSDI*, pages 361–376, Boston, MA, December 2002.

[17] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading transport protocols using mobile code. In *Proc. SOSP*, pages 1–14, Bolton Landing, NY, Oct. 2003.

[18] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

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

[20] SCTP.org. Stream Control Transmission Protocol. `http://www.sctp.org/`.

[21] A. C. Snoeren, D. G. Anderson, and H. Balakrishnan. Fine-grained failover using connection migration. In *Proc. 3rd USENIX Symp, on Internet Technologies and Systems*, pages 221–232, San Francisco, CA, 2001.

[22] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. MobiCom*, pages 155–166, Boston, MA, Aug. 2000.

[23] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computer Systems*, 3(3):204–226, Aug. 1985.

[24] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Connection migration for service continuity in the Internet. In *Proc. 22nd Intl. Conf. on Distributed Computing Systems*, pages 469–470, Vienna, July 2002.