Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions*

Florin Sultan, Aniruddha Bohra
Department of Computer Science
Rutgers University,
Piscataway, NJ 08854-8019
{*sultan, bohra*}*@cs.rutgers.edu*

Liviu Iftode
Department of Computer Science
University of Maryland,
College Park, MD 20742
iftode@cs.umd.edu

Abstract

We propose service continuations (SC), an OS mechanism that supports seamless dynamic migration of Internet service sessions between cooperating multi-process servers. Service continuations provide a server application with a simple and easy to use abstraction, and a means to migrate the service state along with the serviced connection. SC supports transparent resumption of service to the client at another server, and guarantees integrity and consistency of communication channels used by server processes. SC is a generic, application independent mechanism that can be used to provide service continuity and availability for today's complex Internet services.

We have implemented SC in FreeBSD and used them successfully in three real servers: the Apache web server, the PostgreSQL transactional database server, and the Icecast streaming server. We present results of an experimental evaluation showing that using SC adds negligible run-time overhead to existing servers and that SC enable efficient dynamic migration of client sessions.

1 Introduction

The growth of the Internet has led to increased demands by its users with respect to both availability and quality of services delivered over an internetwork where best-effort service is the norm. Critical applications, as well as applications requiring long-term connectivity run over the Internet. In addition, increased user expectancy 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 *end-to-end* service availability.

A vast majority of today's Internet services are built over TCP [17], the standard reliable, connection-oriented Internet transport layer protocol. The connection-oriented nature of TCP, along with its endpoint naming scheme based on network layer (IP) addresses, creates an implicit *binding* between a service and the IP address of a server providing it, throughout the lifetime of a client connection. This makes the client prone to all adverse conditions that may affect the server endpoint or the internetwork, *after* the connection is established: congestion or failure in the network, server overload, failure or DoS attack. With TCP/IP, availability of a service is constrained not only by the availability of a given server, but also by that of the routing path(s) to the server.

Service continuity can be defined as the uninterrupted delivery of a service, from an end user's perspective [20, 27]. The static service-server binding enforced by TCP limits its ability to provide continuous service to the client in the presence of adverse conditions. The only way TCP reacts to lost or delayed packets is by retransmissions targeting the same server endpoint of the connection (bound to a specific IP address) and which cannot exploit the existence of alternate, equivalent servers. In practice, however, the end user of an application with long-lived or critical connections may be more interested in receiving continuous service rather than staying connected to a given server.

Simple server replication does not address the problem. Network failure or congestion after the connection is established may render the service unavailable to the end user. Studies that quantify the effects of network stability and route availability [12, 8] demonstrate that they can significantly reduce the end-to-end availability of Internet services. Although highly available *servers* can be deployed, deploying highly available *services* remains a problem due to connectivity failures.

As server identity tends to become less important than the service provided, it may be desirable for a client to be able to switch between servers during its service session, for example when a server cannot sustain the service. In [27], we have proposed the *cooperative service model*, along with an enabling connection migration protocol, Migratory TCP (M-TCP). In this model, a pool of equivalent servers, possibly geographically distributed across the Internet, cooperate in sustaining the service by handling client connections migrated within the pool. The control traffic between servers, needed to support migrated connections, can be

*This work is supported in part by the National Science Foundation under NSF CCR-0133366 and ANI-0121416

carried either over the Internet or over a private network, different from the one over which clients access the service.

In this paper, we propose the idea of *service continuations* (SC), a new OS-based mechanism to support dynamic migration of live client sessions between multi-process cooperative servers in the Internet. Migration of a session is dynamic in the sense that it may occur multiple times, at any point during session lifetime, transparent to the client and asynchronously with respect to server execution. The SC concept is rooted in the (most often valid) assumption that a service maintains for a client session well-defined *fine-grained* state, and that client sessions are independent of each other.

To the server application, a service continuation represents an abstraction of *discrete* application-level state associated with a client session, spanning multiple process contexts, which is guaranteed to be restored at a new server upon migration. At the OS level, a service continuation is an ordered sequence of fine-grained state components associated with processes involved in servicing the client.

For each process, and for a given client, an SC stores client session state and OS state of communication channels. The first component in the SC sequence corresponds to the front-end process that accepts the client connection for service. Subsequent components correspond to other back-end processes (if any) that participate in servicing the client. To resume the service at a new server, the SC is migrated and used to reinstate the service state and the communication state for all processes involved in the service. We emphasize that migrating a SC does not involve whole process contexts, but only "small" state components associated with a client session.

SC *do not require client applications to change*, while imposing minimal changes on existing server applications. However, both the client and server OS must include support for transparent SC migration. To our best knowledge, SC is the first OS-based solution that provides generic, fine-grained migration support for client service sessions in multi-process servers.

In this paper, we also describe an SC implementation. We use the Migratory TCP (M-TCP) protocol previously designed [26, 23] to migrate and reinstate the client connection (the first channel in the SC sequence). M-TCP uses a limited form of log-based rollback recovery [10] to preserve exactly-once delivery semantics on the migrated connection, without freezing or otherwise disrupting traffic on the original connection during migration. With M-TCP, a decision on when to migrate is taken according to a migration policy that defines triggers specific to the client or to the server. Definition and evaluation of migration policies is out of the scope of this paper.

The goal of this paper is to demonstrate that SC is a viable step towards building highly-available complex Internet services by moving active client sessions between servers. We report on our SC implementation in the FreeBSD kernel. We present results showing that using SC imposes little migration-free run-time overhead on a server, while migration costs are low.

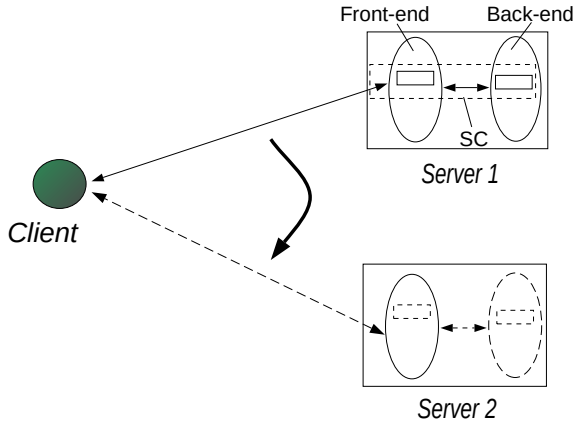

Figure 1: *Cooperative 2-process servers using a service continuation (SC) to support migration of a client session.*

To demonstrate the use of SC in a wide range of real Internet services, we have integrated session migration in three real-world servers. We modified three representative open source servers (Apache [2], PostgreSQL [18], and Icecast [11]) to use SC and migrate client connections during service. The programming effort of using the SC API was low, the major burden of integration being understanding of the server code.

The remainder of the paper is structured as follows. We start with an example in Section 2. Section 3 presents the SC system design. Section 4 describes the mechanism used in SC for synchronization of state components. Section 5 discusses issues related to using SC in building migratory Internet services. Section 6 describes our SC implementation. Section 7 presents an experimental evaluation of the implementation. Section 8 outlines future directions of research. Section 9 reviews related work. Section 10 concludes the paper.

2 Example

We illustrate the SC idea and its use for migration of client service sessions by an example characteristic of today's Internet services. Consider a web-based e-commerce service structured in a two-tier server architecture (Figure 1). A front-end FE (web server) receives a client request, and a back-end *BE* translates it into a sequence of one or more database transactions. The two processes communicate over IPC channels (e.g., pipes): FE parses the request and passes its arguments to *BE*, which executes each transaction in the sequence and sends the results back to FE. The FE forwards the data to the client.

Most of the time, a client may complete its transactions on the server to which it initially connected. However, in case of network congestion or server overload, a client

may experience large delays. When this happens, instead of cancelling the request, the client session can be dynamically migrated to another server to complete the request. The migration must be transparent to the client application (e.g., a web browser), so that the user does not have to restart the whole request.

Note, however, that the client request cannot be re-executed from the beginning at the new server for two reasons. First, this is not possible since the request string is no longer available for the new front-end. Second, restarting execution at the new back-end (if possible) would be wrong, as it may cause already committed transactions to execute twice, compromising correctness. In our example, application semantics require deterministic execution of the sequence of transactions across migration.

With SC, the session can migrate at any time, without synchronizing the FE and BE processes. In order for this to happen: *(i)* the front-end saves the original request in an SC, to use when resuming service at the new server; *(ii)* the back-end saves in the SC the sequence number of the last executed transaction; *(iii)* the OS includes in the SC the state of the pipes connecting FE and BE.

To continue execution after the connection migrates to the new server, the system migrates the SC, passes the migrated connection to the destination FE, and reinstates the state of the pipes connecting the new process pair now servicing the client. Using the migrated SC, the new FE and BE processes restore state pertaining to the migrated client session (the request and the sequence number of the last executed transaction, respectively), and resume execution. The FE passes the request arguments to BE, and BE resumes service by executing the next transaction in the sequence.

An SC must guarantee exactly-once communication semantics across migration for the new pair of processes and must assist it in providing consistent service to client. In addition, session migration needs a transport protocol that supports dynamic migration of a live connection between multiple IP addresses.

3 Service Continuations

In this section, we describe our system model and introduce the SC idea, present the SC migration API and its service guarantees, and provide some background on Migratory TCP [26, 23], the connection migration protocol we use with our SC implementation.

3.1 System Model

Essential to the service continuation idea is the assumption that the state of a server application can be logically partitioned among the clients it services, such that there exists a well-defined, *fine-grained state* associated with each client. As a consequence, the point reached in servicing a given client session can be defined independently of other concurrent sessions. To migrate a live client session, we assume that there exists a *connection migration protocol* that can be used to transfer the associated state between the kernels of the origin (old) and destination (new) server hosts. Any other non-specific state needed to resume service on a migrated session is deemed accessible at the new server.

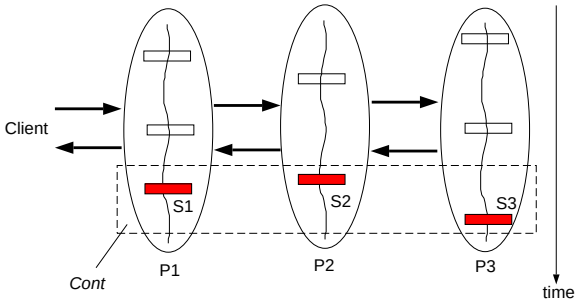

Figure 2: *A process service set and the application view of a service continuation spanning it.*

The per-client server application state may span multiple communicating processes in a *process service set* executing work for the client. This computation model is described by Figure 2, where the process service set $\{P_1, P_2, P_3\}$ handles the service session of one client. Only a select process in the set, called the root or accepting process (P_1 in Figure 2), communicates directly with the client. The other processes can be either workers (processes spawned on behalf of and dedicated to servicing one client) or servers (processes that service multiple clients concurrently). The split-state assumption applies to any server process in the service set. In particular, the state of the root process can be logically split among its incoming connections. Every process has a well-defined and reproducible initial state with respect to a client.

Processes in a process service set communicate via IPC channels (e.g., pipes, fifos, etc.). We assume a uniform abstraction of communication channels (inter-process and client-server) as reliable byte streams.

We assume that execution of each process in the process service set *with respect to a client* can be modeled as a contiguous sequence of intervals delimited by well-defined states (represented as rectangles in Figure 2). Process execution within an interval can be either deterministic or nondeterministic. In a deterministic interval, changes of per-session state in a process are determined only by the data it receives over its incoming communication channels. When a process executes in a deterministic interval, it will always produce the same stream of data on the outgoing channels, given that it receives the same stream of data on its incoming channels.

3.2 Per-Client Service Continuation

For the server application, a *service continuation* (SC) is a *cut* in the global execution state that maps into the

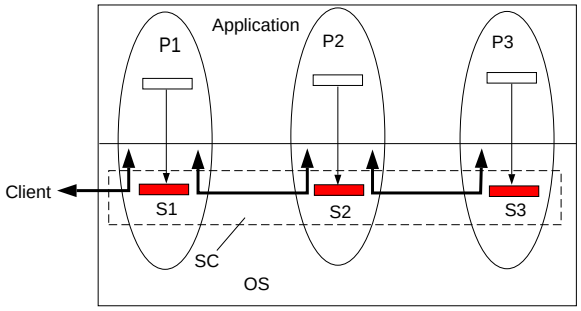

Figure 3: *The operating system view of a service continuation: application saved continuations and the state of communication channels.*

most recent well-defined states reached in servicing a client session by each individual process in the service set. For example, in Figure 2, $Cont = \{S_1, S_2, S_3\}$ is an SC with three per-process components, each describing the point reached by that process while servicing the client. Each component can be individually and independently used by its respective process to resume its service to client. In the OS, an SC is reflected as an ordered set[1] of individual per-process state snapshots, whose content is opaque to the system. A process in the service set would declare a continuation point by saving a state snapshot in the system. Note that state snapshots in an SC are *not* full checkpoints of a process and do not include processor execution context.

Resuming service from an SC involves more than the simple transfer of session state between two hosts. We identify two requirements from the OS support for service continuations. First, in the operating system and at the transport protocol level, a user-level service continuation must be associated with the state of the communication channels (inter-process and client-server). Figure 3 describes the OS view of a service continuation, which includes the continuation points saved by processes in the service set, along with the communication state. A service continuation needs OS support for reinstating the state of communication channels at the destination host.

Second, the OS must perform synchronization of SC state. Note that our model does not impose any form of synchronization on the server processes, nor between the root process and the client. Declaring and saving a continuation point, as well as resuming from it, are operations performed independently by each process in the service set. Note also that the model does not impose any restriction on when a migration may occur, i.e., migration is completely asynchronous with respect to server execution. This requires the OS support to synchronize the two endpoints of a communication channel when resuming service from an SC.

[1]For presentation purposes, the process service set is represented as a chain, although in practice the set might be organized as a tree. In this case, there exists a total order, e.g., given by a tree traversal algorithm on the set, that can be enforced by the system.

3.3 Migration API

SC provides a minimal API that allows a server application to enable migration of a client session by establishing continuation points in any process in its service set. A process must export/import a *snapshot* of application state associated with a client to/from an SC object. The state snapshot must completely describe the point the process has reached in the ongoing service session, so that it can be used as a restart point after a migration. The exported state is opaque to the OS and to the underlying migration protocol.

The SC service interface can be best described as a *contract* between an application process and the system. According to this contract, a process must execute the following actions: *(i)* upon starting service to a client, associate with the service continuation all the communication channels it needs to be restored after migration; *(ii)* export state snapshots during service; *(iii)* import the last state snapshot at the new server after a migration and resume service to the client. In exchange, the system: *(i)* transfers the state of the service continuation to the new server, and *(ii)* synchronizes the state of processes in the service set with the state of their associated communication channels.

The main primitives of the SC API are:

```
cont = create_cont(conn)
export_state(cont, state_snap)
import_state(cont, state_snap)
associate(cont, ch_set)
cont = open_cont(ch)
```

where `cont` is the SC object associated with a client (an OS-specific identifier), `conn` is the client connection in the root process, `ch` identifies an IPC channel, and `state_snap` is an application memory buffer summarizing the client session state in a process.

A root process creates an SC using `create_cont` on the accepted client connection. Processes in the service set use `export_state` to save state snapshots to an SC and `import_state` to retrieve them after a migration. The `associate` primitive is used by a process in the service set to enable service continuation support by the OS for a selected set `ch_set` of its communication channels. The `open_cont` primitive returns the SC object to which channel `ch` was previously associated.

A process must associate channels to an SC, for example before passing or connecting them to other processes and starting communication. The SC object can be either passed between processes, or a process may query an existing channel using `open_cont` to retrieve the SC object it was associated with, then use it to associate its own channels to that SC. The `associate` and `open_cont` calls transitively propagate channel membership in the SC, starting from the root process down the chain, and allocate system resources for stateful channels that must be restored after a migration.

Figure 4 shows the pseudocode for the example in Sec-

```
/* Front-end process (parent) */
while (conn = accept()) {
  sc = create_cont(conn)
  if (import_state(sc, req) == NULL) {
      receive(conn, req)
      export_state(sc, req)
  }

  args = parse(req)
  pipe = create_pipe()
  associate(sc, pipe)
  if (fork() == 0)
      exec(backend, args)
  else
      while (read(pipe, buff) != EOF)
        send(conn, buff)
}
```

```
/* Back-end process (child) */
sc = open_cont(pipe)
if (import_state(sc, {last, tid}) == NULL) {
    connectDB()
    last = 0
}
else {
    status = reconnectDB(tid)
    write(pipe, status)
}
goto last + 1
1:  tid = txn_begin() /* start transaction 1 */
    ...  do work for txn 1
    export_state(sc, {++last, tid})
    status = txn_commit()
    write(pipe, status)
2:  tid = txn_begin() /* start transaction 2 */
    ...
```

Figure 4: *Pseudocode for front-end (left) and back-end (right) server processes in the Section 2 example.*

tion 2, structured as a parent-child process pair, with SC API calls highlighted. The session state consists of: *(i)* front-end: the request string `req`; *(ii)* back-end: the sequence number `last` and the transaction identifier `tid` of the transaction about to commit. The front-end exports `req` to an SC to ensure the original client request is preserved across migration(s). The back-end exports {`last, tid`} to the SC to ensure correct resumption of execution after migration.

For convenience, we assume that the state of a transaction in the DB system can be obtained at any server by calling reconnectDB() with the transaction identifier `tid`.

Note that the same code runs at all servers. If the connection is with the original server, `import_state` returns `NULL` and the session state is initialized locally. If the connection is with a different server (due to migration) `import_state` retrieves the state exported to the SC at a previous server. For example, the request string is read from the connection at the initial front-end and retrieved from the SC after migration to another server. (All SC and IPC calls may fail after a migration; error checking is omitted for clarity.)

If the application follows the terms of the SC API contract, the system supports service resumption in all processes in a process service set and guarantees integrity and consistency of their data streams. The migration API decouples the server application logic from the actual migration time by enabling asynchronous migration, makes the scheme lightweight and yields good performance.

3.4 Connection Migration Protocol

To enable SC, a connection migration protocol is needed as carrier of SC-encapsulated state of a client session. The protocol must also support transparent resumption of communication on a migrated client connection.

To prototype our system, we have used Migratory TCP (M-TCP) [26, 23], a protocol previously designed by us that supports connection migration in single-process server instances. In addition, the protocol enables a stateful server to resume execution by transferring an application-controlled amount of specific state with the migrated connection.

M-TCP provides enabling mechanisms for the cooperative service model of Figure 1, in which an Internet service is represented by a set of (geographically dispersed) equivalent servers. A client connects to one of the servers using a reliable connection with byte-stream delivery semantics. The complete client interaction with the Internet service from the initial connect to termination represents a service session. M-TCP enables the session to be serviced by different servers through transparent connection migration. Connection migration involves only one endpoint (the server side), while the other endpoint (the client) is fixed. The M-TCP protocol layers at the old and the new server cooperate to facilitate connection migration by transferring endpoint state.

The mechanism for connection migration in M-TCP reincarnates the migrating endpoint of the connection at the destination server. The mechanism can also be used to establish a restart point for the server application by transferring supporting SC state. Depending on the implementation, the transfer of state can be either *(i)* on-demand, i.e., at the time migration is initiated, or *(ii)* proactive, i.e., in anticipation of migration.

Figure 5 describes the sequence of steps in a migration. Initially, the client contacts the service through a connection C_{id} to a server S_1. During connection setup, S_1 supplies the addresses of its cooperating servers, along with migration certificates. At some later point during connection's lifetime, the client-side M-TCP may initiate migration of C_{id} by opening a new connection to one of the cooperating

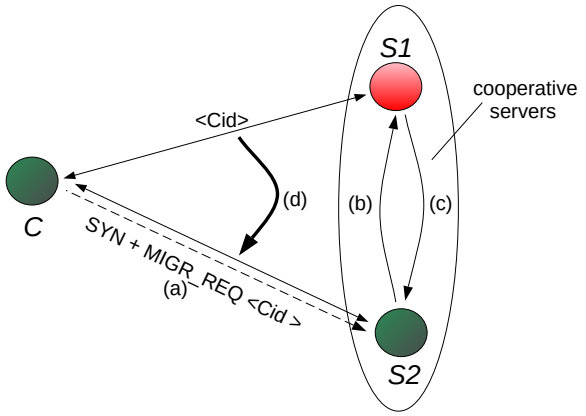

Figure 5: *Migration mechanism in M-TCP. Connection C_{id} migrates from S_1 to alternate server S_2.*

servers (S_2), sending the migration certificate in a special option (Figure 5 (a)). To reincarnate C_{id} at S_2, M-TCP transfers associated state from S_1. Figure 5 shows the on-demand transfer version: S_2 sends a request (b) to S_1 and receives the state (c). If the migrating endpoint is reinstated successfully at S_2, then C and S_2 complete the handshake, the new connection takes over and S_1 drops its endpoint (d).

During the migration handshake, the client, the destination and the origin server exchange information (sequence numbers, buffered unacknowledged segments, etc.) to synchronize the new endpoints of the connection. M-TCP is heavily optimized to minimize the amount of data (protocol specific state) transferred between servers during a migration. The protocol overlaps the migration with data transfer on the original connection in order to reduce its impact on the client application.

When migration is initiated, the C endpoint of the original connection enters a `MIGRATING` state, in which M-TCP continues to receive, acknowledge and deliver to the client application any data that may still arrive from S_1. While the client application is still able to do I/O operations on its connection endpoint, the `MIGRATING` state blocks the upstream data flow from C to S_1, as it would waste resources to send data to a server from which the connection is moving away.

If migration to S_2 fails, the original connection can either be seamlessly restored to its previous state and continue with S_1 as an endpoint, or it can continue trying to migrate to another server. If migration succeeds, the client-side endpoint of the old connection is forced into a special `BLACKHOLE` state. Its role is similar to that of TCP's `TIME_WAIT` state, with the exception that protocol output from this state is completely suppressed.

Migration of a synchronized connection in M-TCP is possible in any state of its endpoints except for the `TIME_WAIT` state, may occur many times and at any point during connection lifetime without disrupting the ongoing traffic, and is totally transparent to the client application.

4 SC Synchronization

One of the salient features of the SC abstraction is that it does not enforce synchronization on the process service set or between the client and the root process. In this section, we describe the mechanism used by the OS service continuation support for synchronizing the per-session state of a process and the state of its communication channels. For this we assume, without loss of generality, that a communication channel can be modeled as a unidirectional reliable byte stream between a writer W and a reader R. The client-server connection itself can be viewed as a pair of such streams, hiding TCP's reliability mechanisms.

In order to resume service to a client at a new server, R and W import state snapshots from an SC. Since the corresponding processes at the old server did not synchronize when exporting their snapshots, and because migration is asynchronous with respect to process execution, R and W will not be synchronized when resuming service. Similarly, while executing, the in-kernel data buffering and the active read/write communication cause the state of a channel at a given moment in time to go out of sync with respect to the state reached by R and W as a result of reading/writing data from/to the channel.

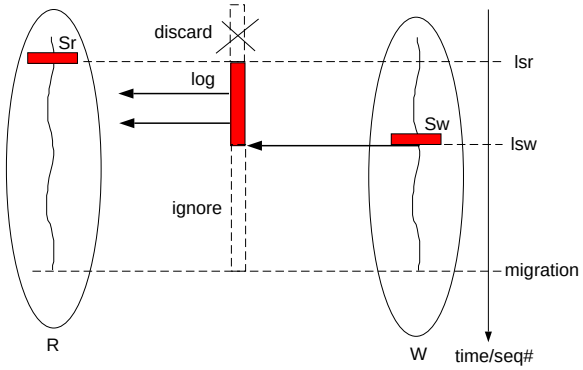

Figure 6: *Synchronization with service continuations. Reader behind writer.*

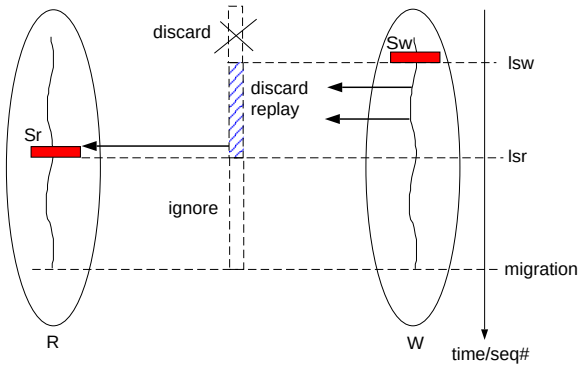

Figure 7: *Synchronization with service continuations. Reader ahead of writer.*

Figure 6 shows an example of execution where reader R is behind the writer W in taking its state snapshot. The

vertical bar represents a virtual infinite buffer abstracting the channel over which the two processes communicate. For convenience, we use as timeline for events in a process the sequence number of the next byte of interest in the buffer. For example, the moment at which a process takes its snapshot is marked by the sequence number of the first byte it will read/write after the snapshot. For R, lsr denotes the first byte to read after taking its last snapshot S_r. Similarly, lsw denotes the first byte to be written by W after its last snapshot S_w. In Figure 6, since $lsr < lsw$, upon resuming service at the new server, R will issue reads for data that can no longer be supplied by W.

To solve the problem of synchronization between application-level state and the communication state in the OS, we use a *limited* form of log-based rollback recovery [10] to restore the session state in a process at the new server. Upon restarting service for a migrated client at the new server, a process in the service set imports the last state snapshot from the SC and uses it to initialize its local session state. After resuming execution, the process may *replay* execution already done at the old node since the snapshot. This includes reading data that was read at the old node, and writing data that was already written.

To support the replay of data read from a communication channel, the system must *log* and transfer from the old node data that cannot be generated by the writer at the new node. In Figure 6, the synchronization log consists of only the $[lsr, lsw)$ region of the buffer. Data at sequence numbers less than lsr can be discarded as it will not be needed by R after restart. Data in the interval $[lsw, migration]$ can be ignored by the system (not included in the SC state), since W's replay will regenerate it after restart.

Figure 7 shows the case when the reader is ahead of the writer in taking its snapshot, i.e., $lsr > lsw$. In this case, during replay, R will read only from sequence numbers larger than lsr. These can be regenerated by W's replay, and thus ignored by the system. When R takes its snapshot at lsr, the system discards any previous log. During replay, the system will discard any write issued by W in the interval $[lsw, lsr)$.

This model can be easily mapped into the TCP connection migration protocol used between client and server, taking into account the presence of a reliability mechanism. Note that although the client does not take state snapshots, we can emulate its lsr and lsw by considering the data that the *protocol* endpoints at client C and origin server S_1 have received and acknowleged at the moments when migration is initiated and takes place. The two values are computed at S_1, at the time of migration t_{migr}, using the protocol state of the connection endpoints. For client as a reader: $lsr = \max(nxt^C(t_{init}), una^{S_1}(t_{migr}))$. For client as a writer: $lsw = \max(una^C(t_{init}), ack^{S_1}(t_{migr}))$.

Here, $una^E(t)$ is the sequence number of the first unacknowleged byte sent from endpoint E, $nxt^E(t)$ is the sequence number of the next byte expected to be received at E, and $ack^E(t)$ is the sequence number of the next byte to be acknowledged by E, all at time t. t_{init} and t_{migr} are, respectively, the moments when migration is initiated by the client-side connection migration protocol, and when migration takes place and state is transferred from S_1 to S_2.

Because it relies on execution replay, SC synchronization must take into account potential nondeterminism. Recall that the SC model in Section 3 allows nondeterministic intervals (NDI) of process execution. When executing in an NDI, writes from a process must not propagate to client or to other processes, as this may trigger state changes that cannot be reproduced by replay. In Figure 7, suppose that S_w marks the beginning of an NDI. Then, if resuming from S_w after migration, W may issue different writes from those issued at the old server. Note however that R's state has already advanced based on old writes. In effect, when resuming the session at the new server, the S_r and S_w snapshots will be inconsistent and may cause incorrect replay.

The solution is to disable write propagation from an NDI, and re-enable it when writes have become stable with respect to a potential migration, i.e., when the NDI can no longer be replayed. This happens at the moment a new snapshot is recorded in the SC. With this observation, the problem can be easily solved by annotating NDIs. For example, with a simple extension of the API, `export_state` can include a flag to declare the type of interval the process will enter after export. If the interval is an NDI, the OS will block write propagation on output channels until the next export operation.

The most recent service continuation, including application-level continuations as state snapshots and the communication state, allows any process in the service set to: *(i)* restart servicing the client session from the state snapshot on, *(ii)* replay data read at the old server that cannot be supplied (regenerated) by a writer process or by the client, and *(iii)* supply data it had produced at the old server before the last snapshot (i.e., data that it cannot regenerate by execution replay at the new node). The above three guarantees, coupled with piece-wise deterministic execution and controlled propagation of writes issued in nondeterministic intervals, ensure that the new server can resume service to the client consistently after migration.

5 Building Services with SC

5.1 SC and Nondeterministic Execution

The execution model adopted by SC (Section 3) assumes that there exist deterministic execution intervals in a process during which the only *external* cause of changes in the state of a given session is the contents of its byte-stream input channels. This model limits the amount of nondeterminism in an application in order to achieve a *practical* solution to what essentially is a distributed recovery problem [10]. SC exploits it in its synchronization scheme.

In distributed consistent recovery protocols, communica-

tion channels are a major source of nondeterminism. To deal with it, processes either coordinate during normal operation to establish a consistent recovery line, or compute it during recovery. None of these is a feasible solution in the case of Internet service sessions, as we do not want to force coordination on processes and, moreover, we have no control over those running on the client side. By replaying and generating (within deterministic intervals) the same sequence of bytes[2] on input/output channels before and after migration, an SC ensures deterministic behavior with respect to a client throughout its process service set and eliminates reads from communication channels as a source of nondeterminism.

The remaining *known* sources of nondeterminism in the execution of a process reside in all the other synchronous interactions with the kernel (system calls). They can be eliminated if the application can ensure that a nondeterministic change in state does not affect the environment of a process (including its output channels), so ultimately it is not made visible to the client. This can be achieved using application knowledge in two ways: *(i) annotate* nondeterministic intervals as described before (the system will block output propagation from such intervals until they are closed), or *(ii)* take a snapshot before committing a nondeterministic change of state (i.e., before propagating its effects on output channels) and include it in the snapshot. For example, if the server must send to a client the result of a `gettimeofday()` call, it must take a snapshot before the send, recording the result in the snapshot. If the session migrates to another server, then, after restart from the snapshot, the new server must send the value imported from the snapshot instead of executing `gettimeofday()` locally.

Note that our model cannot account for purely asynchronous events like signals. Replay of asynchronous signals is a hard problem in itself which was not directly solved even in dedicated fault-tolerant systems like [7]. Such systems achieve accurate signal replay in two steps: *(i)* convert signals to messages sent over *signal channels* and log them at dedicated backup processes; *(ii)* keep a primary-backup process pair strictly synchronized at the time a signal is delivered, so that a restart of the backup always begins with a signal delivery event. A similar scheme could be used with SC if the system can take forced snapshots in a process, which may not be possible in general. What we essentially achieve with the SC synchronization scheme in the general case is a reasonable tradeoff between controlled nondeterminism and system complexity.

5.2 Migration Policies

SC provide only the mechanism to support migration of live instances of service sessions. Definition and evaluation of migration policies is beyond the scope of this paper. In [26] we proposed, in the context of M-TCP, a migration architecture that decouples a migration mechanism from policy decisions, including the events that trigger a migration. This allows various migration policies to be designed and evaluated independently. In this architecture, migration triggers can be placed: *(i)* at the client side, to dynamically switch to another server if the current server becomes unavailable or if it does not provide a satisfactory level of service; *(ii)* at the server side, under control of the server application, to implement a load balancing scheme by shedding some of the connections to other less loaded servers, or to implement an internal policy on content distribution among groups of clients. In one of the experiments in Section 7 we use a sample policy designed for experimental purposes which can also be applied in practice.

5.3 Application Classes

Under the cooperative service model, SC can be used to react to adverse conditions that hamper service availability and/or quality of service received by clients. Such conditions include server overload, network failure or congestion on a given path, etc. In these cases, migrating the session to another server ensures sustained service to the client.

To benefit from migration, an application must be willing to incur a cost, which should be amortized in terms of either better service quality or increased service availability over the lifetime of the connection. For example, it does not make sense to enable migration for short lived connections when the service is not critical to the client. We identify two classes of services that can benefit from SC support:

(i) Applications that use long-lived reliable connections. Examples are multimedia streaming services [11], applications in the Internet core that use TCP communication over large spans of time [19], etc. In Section 7, we report on our experience with such applications.

(ii) Critical applications. Characteristic to this class is that users expect both correctness and good response time from the service. Examples are Internet banking and e-commerce. In [25] we made a detailed case study of integrated system support for migration in a transactional database system [18] accessed over the Internet, that can be used in this class of applications.

5.4 SC Programming Tradeoffs

With the SC API, a server application may have to exploit certain performance tradeoffs. One tradeoff is between the size of the state snapshot and the amount of SC state in the system. Delaying export of a snapshot for too long in order to optimize its size may increase the amount of logs. In general, a reader from a "fat" communication channel should take snapshots frequently to allow the system to discard logs it maintains for it. In Section 6 we describe an efficient implementation of the export primitive that practically eliminates the overhead of frequent snapshots.

Another tradeoff is between the snapshot frequency and the

[2] For the purpose of the SC synchronization scheme itself, enforcing the same *number* of bytes would suffice.

amount of work redone at the new server after migration: a larger frequency may mean more overhead but less work to be redone. Increasing the snapshot size may, in some cases, capture more computation and might result in less work being redone after restarting at the new server. On the other hand, larger snapshots may increase the run-time overhead and impact the migration time.

Programming with SC requires an effort in understanding the SC model (Section 3) and defining computation of a server process on behalf of a client as a sequence of state intervals. Most existing servers work in this way. Still, programming errors may occur if application logic does not obey the SC model, as illustrated by the following example. Suppose a process servicing a client reads a chunk *A* of data from a channel and, before consuming it, it immediately exports to an SC a snapshot *S* that does *not* reflect a change in the client session state as a result of reading *A*. If a migration occurs right after the export, the corresponding process at the new server will resume service from *S*. If the process naively attempts to "read" *A* now, will instead receive the next chunk in sequence. While it may appear as if chunk *A* has been "lost," this behavior is in fact normal. The OS discarded *A* from the channel log at the time *S* was exported, as *A* had been read before and the OS assumes that *S* incorporates changes in session state caused by *A*. This simple example exposes in fact a programming error in using the SC. A correct program must not attempt/expect to "read" data it has read before exporting a snapshot since, according to the state-driven execution model of Section 3, the state snapshot must reflect the fact that the read has already been performed. Because an `export_state` call aggressively discards logs from the system, data that has been read from a channel is not "safe" until written on another channel or reflected in a state snapshot.

However, it is possible that a process does not change the *session* state based on data it reads from a channel. As an example, consider a process whose only task is to forward data in a process chain by reading chunk *A* from an input channel and immediately writing it to an output channel. While apparently the process does no explicit computation based on *A*, its state with respect to the client session *does* change after the read, as it now includes the raw data in *A*. For this reason, the process must ensure that, at the time it does an export, it has forwarded on its output channels all data it has previously read. If not, it has to save this data in an exported snapshot to avoid losing it in case of migration. In conclusion, a forwarding process must not take a snapshot with buffered data, as this data is *volatile* with respect to migration between the read from a channel and its forwarding on another channel.

With SC, a server application must obey a certain programming discipline, using our API to export/import to/from the system state associated with a potentially migrating client session. The client application is not required to change. From our experience with three real, widely-used server applications described in Section 7, we believe that the programming effort involved in using SC should be fairly low, and expect the API not to be very intrusive to application logic. Adhering to a certain programming discipline and API is the effort a server writer may want to invest in order to take advantage of dynamic server-side migration of live client sessions.

6 Implementation

We have implemented service continuations in the FreeBSD 4.3 kernel, including support for migrating TCP sockets and OS pipes. We use M-TCP as the underlying connection migration protocol. Our M-TCP implementation is compatible and inter-operates with the existing TCP/IP stack. The protocol works as an extension to TCP, with M-TCP control information sent as TCP options, thus enabling coexistence of hosts that are M-TCP capable with those which are not. At connection setup time, the client and the server side negotiate their capabilities in terms of support for connection migration. If either of the parties does not run M-TCP, then the connection defaults to a regular TCP connection. For state transfer between two server hosts, M-TCP uses a TCP connection established between the kernels of the two machines, possibly over a dedicated network.

In terms of overhead, supporting SC requires *time* (spent for export operations and logging data in the system) and *memory* (used to store SC state). Logging is essentially zero-cost in terms of time, being done in-place in already existing kernel buffers, on every write operation. However, since logs are kept in system memory, their size must be limited.

An important performance issue with the SC API is that a server process must cooperate with the system and export state snapshots. As seen in Section 4, an export operation by a reader is beneficial to the system in two ways: it discards logs maintained in the system, and reduces the amount of state to be transferred to a destination server during migration. In general, an export operation may also benefit the application as it reduces the amount of work to be re-executed after migration. However, to the application, frequent exports mean run-time overhead during migration-free execution. We thus have two conflicting requirements: for the system, frequent exports are desirable to reduce resource usage; for the application, they may incur high overhead during migration-free execution. This means that the `export_state` primitive must be carefully implemented.

One way to implement the export primitive is *eager export*, by copying a snapshot in the SC on every call. Eager export suffers from the unavoidable overhead of copying state from user to kernel space, which may become significant if snapshots have a large size and their frequency is high.

An alternative implementation called *lazy export* is possible, based on two observations. First, in certain cases, snapshots may represent only soft state in a service continuation. If such a soft snapshot is ignored or delayed,

the performance of the system might be affected (more resources are used and more work is lost by restart from an older snapshot). However, the correctness of the system is not compromised. In the lazy export implementation, we choose to defer *recording* a snapshot until the following read/write operation of the process on a communication channel, thereby saving an extra system call.

A second observation is that the actual copying of the snapshot into the SC can be further delayed until it is actually needed, i.e., until migration time. To defer recording and copying a snapshot, the process must use a special `register` call to register a pair of alternating snapshot buffers with the SC. The `register` call pins the user-level buffers into physical memory and maps them into kernel memory; the system copies the last snapshot into the SC at migration time. When a new snapshot becomes available, the process increments a sequence number in the buffer to help the system identify it as the most recent. A pair of buffers is needed since migration may occur asynchronously and the snapshot copied into the SC while its buffer is being modified in the application. With these optimizations, the cost of lazy export reduces to a one-time `register` operation. Experiments with real servers under load show that pinning and mapping snapshot buffers adds little system memory overhead.

Our implementation supports both variants of export and their arbitrary mix in an application program. A performance comparison is presented in Section 7.

7 Experimental Evaluation

7.1 Microbenchmarks

The goal of our evaluation is to show that SC can be implemented efficiently and has little impact on application performance. We perform three experiments to estimate the migration costs, the migration-free run-time overhead of the SC API, and the impact of using SC on performance metrics of interest to a client.

In the first experiment, we use a microbenchmark application to measure the time taken for migration and its breakdown into various components. This quantifies the responsiveness of the system to migration triggers. In the second experiment, we modify the TTCP [28] benchmark to use SC on the server (sender) side, and measure the throughput perceived by a client in several migration scenarios. In the third experiment, we use SC in a synthetic streaming server. We employ a client-side rate-based migration policy to achieve good streaming throughput by migrating a client connection when server performance degrades.

The experimental setup consists of two identical Intel Celeron 400MHz PCs as servers and a 233 MHz Pentium II PC as client, all running our modified FreeBSD 4.3 kernel that includes SC and M-TCP. All nodes have 128 MB RAM and are connected by 100 Mb/s Ethernet on an isolated subnet. Server hosts are interconnected through a second 100 Mb/s Ethernet interface dedicated to M-TCP control traffic. The cooperative server applications service client requests on the primary (service) interfaces, while M-TCP uses the secondary (control) interfaces.

7.1.1 Migration Costs

The first experiment estimates the migration costs for one-process and two-process SC, as a function of the SC state size. In the two-process case, server processes are connected by three OS pipes. We use a migration-aware synthetic server application that does not generate data. This eliminates any logging for SC synchronization, and thus variability in the amount of state transferred between servers. At the same time, we control the amount of SC state by conveniently varying the size of the exported snapshots in the server, between 0 and 10 KB. In the two-process SC, the whole state is exported only by the root process. We do not consider other state size distributions across processes, as the dominant parameter in our experiments is the *total* amount of SC state transferred from the old to the new server during migration.

Table 1 and Table 2 show the time breakdown into various components at the client (C), new (S_2) and old (S_1) server, for a single one-way migration, in the single-process and two-process SC, respectively. The components shown are: *(i)* C: time to initiate, wait for completion, and complete a migration; *(ii)* S_2: time to prepare a SC state request, wait for the state, and reinstate the migrated SC; *(iii)* S_1: time to prepare a SC state reply. Times are measured inside the kernel, using the processor cycle counter. To eliminate the impact of the nondeterministic network on the fine-grained measurements, we selected values corresponding to the minimum client Wait time observed over 200 runs.

When a measured interval is opened by the receipt of a message in M-TCP, the value includes the time spent in M-TCP processing (columns 5, 8), i.e., the timer is started when processing enters the transport protocol software. If an interval is closed by the receipt of a message (columns 3, 6), then the measured value includes all associated stack processing. Although this does not account for all the time spent in the protocol stack, it allows us to compare the values against measured ICMP latency for the "empty" and "full" TCP segments characteristic to the M-TCP control traffic in our setup. The average RTTs measured are: C-S link, empty segment: 138 μs; S_1-S_2 link, empty segment: 124 μs; S_1-S_2 link, full segment: 460 μs.

From Tables 1 and 2 we note that the cost of the migration (C Wait) is, as expected, dominated by network latency, mainly due to the SC state transfer from S_1 to S_2 (S_2 Wait reply). The actual time spent in host processing, for manipulation of SC state at the server side and handling migration at the client side, is the sum of columns 2-8 except for the two "Wait" columns. Our in-kernel SC/M-TCP implementation takes only 213 μs and 331 μs at the two servers combined, for the single and two-process SC, respectively, for the largest amount of state transferred

State size [*KB*]	*Client* [μs]			*Server 2* [μs]			*Server 1* [μs]
	Initiate	Wait	Complete	Prepare req.	Wait reply	Reinstate	Prepare reply
0	68	360	6	69	183	17	77
1	71	503	6	71	317	21	86
5	76	980	6	76	789	20	135
10	71	1,590	6	81	1,392	23	132

Table 1: *Breakdown of migration time for one-way migration with a single-process SC.*

State size [*KB*]	*Client* [μs]			*Server 2* [μs]			*Server 1* [μs]
	Initiate	Wait	Complete	Prepare req.	Wait reply	Reinstate	Prepare reply
0	70	485	6	108	250	23	132
1	70	608	6	100	388	22	146
5	70	1,066	6	101	844	22	183
10	73	1,681	6	99	1,458	25	232

Table 2: *Breakdown of migration time for one-way migration with a two-process SC.*

in the experiment. Note that additional time is spent in protocol processing at S_2 but this overlaps with the network latency included in the Wait reply component.

As expected, from Tables 1 and 2 it can be seen that the client Wait time increases almost linearly with the SC state size. Correspondingly, other times that depend on the state size (Prepare reply and Wait reply) increase with it. An anomaly is the decrease of Prepare reply value at 5 and 10 KB in Table 1. We suspect it is due to our choice of the best client-side migration time value, which, combined with network-induced variability, selects values that might be affected by noise at a server (note that, for a given state size, values in columns other than client Wait are *not* minima in their class).

A last observation is that the sum of the three client components in Tables 1 and 2 represents the actual migration time as perceived at the client side. It is important to note that this time is an interval between two *protocol* events, i.e., between the moment at which the client-side protocol software initiates migration to a new server and the moment when data exchange can start with this server. With a connection migration protocol like M-TCP, which heavily overlaps migration with data delivery from the old server, the client-side migration time does *not* necessarily reflect a gap in communication for the client application and therefore may not represent a direct cost to the client.

7.1.2 Migration Overheads

The second experiment evaluates the migration-free run-time overhead of using the SC primitives and the overhead of migration, by measuring their impact on client-perceived performance. We modify the TTCP benchmark [28], using SC on the server side to migrate and resume a data transfer at a new server, from the point where the previous server left off. The performance metric is the effective throughput perceived by a client when receiving 400 MB in a continuous data stream. Throughout a run, the client connection is either stationary (has a fixed server endpoint), or it migrates periodically between two TTCP servers.

We run experiments with single-process and two-process TTCP servers. In the two-process case, the second process writes data over a pipe to the first process which sends it to the client. A server process takes state snapshots using the `export_state` primitive, after every 8 KB of data sent. In the two-process case both processes take snapshots using the *same* variant of the `export_state` primitive, i.e., either eager or lazy (Section 6). The actual state snapshot consists of one integer that represents the position in the stream reached by the server. We pad this up to the state size required by a particular run of the experiment. Similarly to the previous experiment, all the state is exported by the first process, i.e., the second process takes 0-size snapshots.

We first measure the effective throughput sustained by a "base" TTCP server that does not use the SC migration API, therefore does not experience any run-time overhead for migration support. The values observed for the single and two-process base server are 7,988 KB/s and 7,934 KB/s, respectively.

To estimate the migration overheads, we next run two SC-enabled cooperative servers and measure the throughput seen by a stationary client and by clients that migrate between servers at intervals of 2, 5 and 10 seconds, respectively. We run each client for three values of the exported server state size (1, 5, and 10 KB). For a given size, we repeat the experiment with servers that differ in the variant (eager or lazy) of the `export_state` implementation they use. To eliminate network-induced variability, we take the maximum values observed over 200 runs of each configuration. Figures 8 and 9 plot the results for the single-process and two-process server, respectively. Graphs are scaled to emphasize differences otherwise small in absolute value.

The loss in performance from the base server case for a stationary client gives a raw measure of the run-time

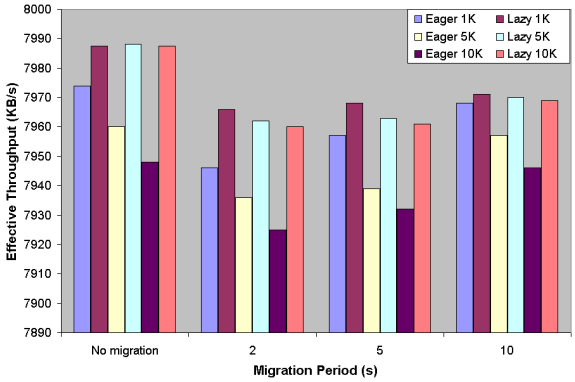

Figure 8: *Performance of a 400 MB TTCP transfer from a single-process SC-enabled server.*

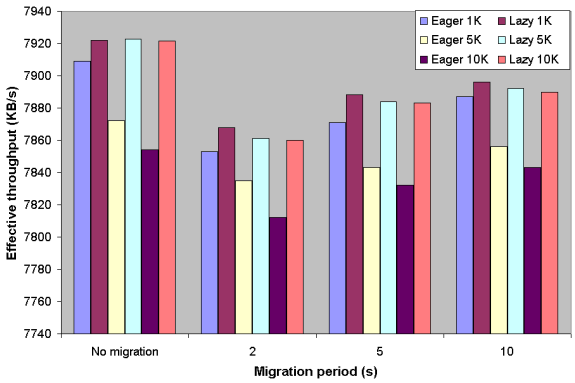

Figure 9: *Performance of a 400 MB TTCP transfer from a two-process SC-enabled server.*

overhead of adding SC to an existing server. For a migratory client, this performance loss is further compounded by migration-induced costs.

In Figure 8, the stationary client has no performance loss with a lazy server, while with an eager server it exhibits an overhead increasing with state size. Across all migratory clients, the largest performance hit (under 1%) is taken by a client with the smallest migration period (2 s), serviced by eager servers with the largest amount of per-client state (10 KB).

In the two-process case in Figure 9, the stationary client has a small drop in performance with a lazy server, regardless of the state size. With an eager server, the loss increases with state size up to 1%. The fact that in the lazy case the loss is not correlated to the state size confirms our expectation that a lazy server should not introduce direct overheads related to taking snapshots (as also seen in the single-process case). However, the *existence* of a loss can be explained by other side-effects of IPC logging, like an increase in the frequency of context switches. Since we do not change the default pipe buffer size of the system, the extra space temporarily occupied by logs in the buffer may cause the writer to block and trigger a context switch for the reader to consume data and free up space. This effect can be eliminated by tuning the default pipe buffer size. The largest performance hit taken by a migratory client in the two-process case is 1.5%,

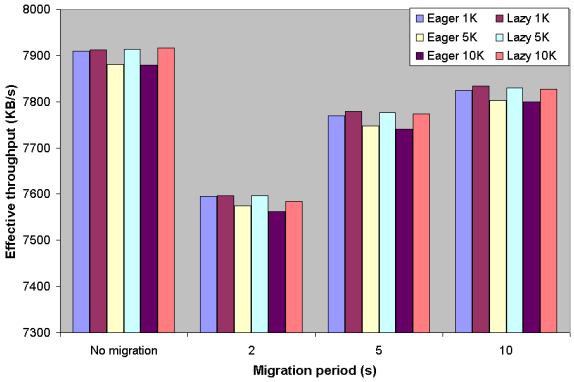

Figure 10: *Performance of the TTCP two-process transfer when the second process does not take snapshots.*

again with the smallest migration period and eager 10K servers.

Several general observations can be made on trends exhibited by the two figures with respect to server type, state size and migration frequency: *(i)* a lazy server does not introduce any run-time overheads related to state size, and introduces no overhead for a single process server; *(ii)* a lazy server performs consistently better than its eager counterpart; *(iii)* the eager server performs worse with increasing state size; *(iv)* for the migratory clients, the migration overhead increases with increasing migration rate, across all state sizes and server types; *(v)* for the same migration rate, the eager server performance decreases more abruptly with state size as compared to that of the lazy server. Since the migration-free performance of a lazy server is unaffected by the state size, this means that, for the cases under study, the migration overhead is relatively smaller than the migration-free overhead of an eager server. All these observations confirm behaviors we expected from the SC design and implementation.

Figure 10 is a particularly interesting case of a TTCP two-process server that demonstrates the impact of write discards during replay at high migration rates. In this experiment, the first process (the pipe reader) takes snapshots, while the second process (the pipe writer) does not. As a result, the SC state of the pipe records that, after a migration and restart, writes by the second process must be discarded before it is allowed to generate new data for the reader (and implicitly for the client). Because the reader does advance its state (the position in the stream) with every snapshot, while the writer always starts its replay from the beginning, more and more data from the second process has to be discarded after each migration. In addition, the lower the migration period, the more time the writer spends doing useless work because its writes are discarded, relative to the time during which it generates new data. Less new data generated from each server translates into an overall drop in throughput with increasing migration rate, visible in Figure 10 when compared against Figure 9. This example demonstrates "bad" application behavior, and makes the point that an application should be concerned with the

State size [*KB*]	*eager export* [μs]	*import* [μs]	*associate* [μs]	*register* [μs]	*unregister* [μs]
1	22	26	150	124	69
5	47	68	151	148	80
10	114	121	154	154	81

Table 3: *Cost of SC system calls for three application state sizes.*

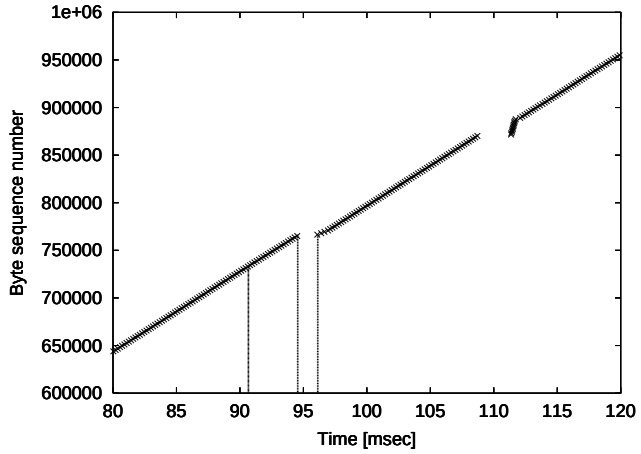

Figure 11: *TTCP transfer trace. The first gap corresponds to a migration.*

length of replay in case of migration. Note that in this case there are no pipe logs since reader's snapshots are always ahead of the nonexistent snapshots of the writer. The server processes should, however, stay "close" enough in taking snapshots, in order to reduce potentially long, useless replays.

Note that we run TTCP as a microbenchmark, so small differences in server run-time overhead between eager and lazy servers may not count too much in this case. However, for a real server with thousands of clients, the difference in implementing an efficient system call would definitely pay off, especially at large state sizes.

For completeness, Table 3 shows the cost of SC primitives, measured as time that a process spends in the corresponding system calls, in the two-process TTCP server. All calls, except for `export`, are one-time operations over the period during which a client stays connected to a given server. Note that the cost of the eager `export` call increases with the state size, while the cost of `register` also increases slightly. Recall, however, from Section 6 that `register` is a helper call used with lazy export, issued only once by a process per serviced client. In contrast, in an eager server, an `export` call is made every time a process must take a snapshot (in our TTCP experiments, about 50,000 such calls are made by a process during a run).

To further understand why a long TTCP transfer suffers a small loss in performance even with frequent migrations, we collected a fine-grained trace of sequence numbers of segments received on the connection from single-process eager 10 KB servers. The trace samples an equal number of segments before and after a migration. Figure 11 shows a detail of the trace, including the migration (first gap) along with another event that disrupts the transfer. Time and sequence numbers are relative to the first recorded segment arrival. The vertical lines mark, in order from left to right: the first segment seen from the old server *after* migration was initiated, the last segment seen from the old server, and the first segment received from the new server after migration. Note the high degree by which M-TCP overlaps the migration with data being received and delivered from the old server.

The second disruption in Figure 11 is caused by a burst of incoming segments which generates a flurry of interrupts from the network interface. This prevents normal input protocol processing by the receiver, until the sender stops sending because the receiver window has filled up. During the burst, the received segments are queued up. When protocol processing resumes, it first consumes all queued segments, proceeding at a high rate (as seen from the steep slope of the trace after the gap), then continues at the steady arrival-driven rate.

Comparison of the two gaps in the trace shows that a disruption due to migration is not much worse than those caused by other events in the system, which means that migration should not be a heavy hit to the client. The low migration costs and the ability of the connection migration protocol to receive data from the old server while migrating the connection explain the overall good performance of the migration.

7.1.3 Sustained Performance

The third experiment tests the sensitivity of an SC migration enabled streaming service to degradation in server performance, when using a rate-based migration policy. We use a synthetic single-process streaming server that sends data to a client at regular intervals as a sequence of chunks of 1 KB. After sending a chunk, the server takes a snapshot (using the eager export primitive) recording the position it has reached in the stream. After sending 32 KB on a connection, we simulate a degradation in server performance by gradually increasing delays between successive chunks. This behavior affects the throughput as perceived by the client. On the client side, an in-kernel migration policy module uses the inbound data rate as metric and triggers migration when the estimated rate drops by 25% from the maximum rate seen on the connection from the current server.

Figure 12 shows the trace of byte sequence numbers observed by the client-side M-TCP on the connection, up to the maximum of 256 KB, where we cut the stream. The graph shows four cases, one without migration and three with migration for SC state snapshot sizes of 2, 10, and 16 KB. The stationary trace exhibits the decaying service profile of a server. In the migratory traces, each slope discontinuity corresponds to a migration, after which data is received at the best rate from the new server. The net result

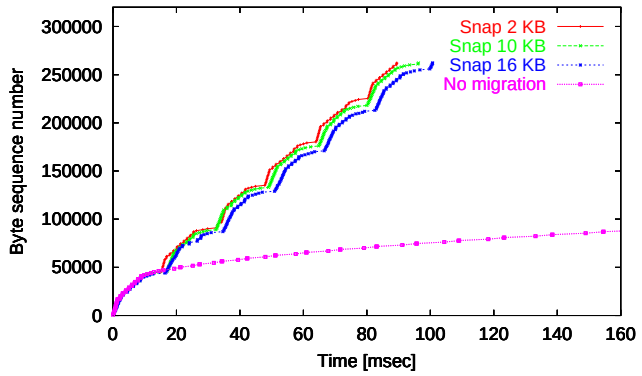

Figure 12: *Traces from a single-process streaming service on stationary and migrating sessions.*

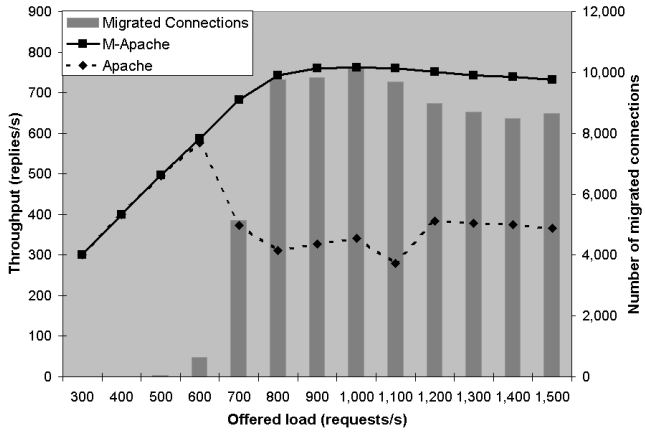

Figure 13: *Apache and M-Apache throughput under load (Rutgers trace). M-Apache migrates a connection if the transfer takes more than 10 s.*

is that the effective rate at which the client receives data is fairly close to the average sending rate of a server before its transmission rate drops sharply. The graph also exhibits the cumulative effect of the time taken by each migration and of the (eager) export overhead, reflected in the longer time it takes the client to receive the stream as the snapshot size increases. The experiment shows that the effective throughput perceived by a client improves by transparently migrating the connection in case the server cannot provide a satisfactory rate, and validates the feasibility of using SC for sustained streaming service.

7.2 Real Applications

Web Server. We have modified the Apache [2] web server to use our SC API, enabling transparent migration of client browser connections between different M-Apache (Migratory Apache) servers. The SC API adds just 50 lines of code to the base Apache and supports migration for transfers of both static and dynamic (CGI) content. A per-client state snapshot in M-Apache consists of the client request and one integer for the file offset reached during transfer. M-Apache exports a state snapshot after every 8 KB sent on a connection and after having sent a whole reply.

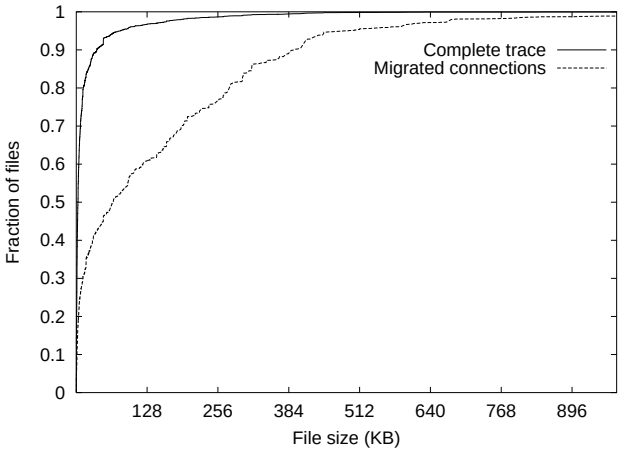

Figure 14: *CDF of file sizes for the complete trace and for the connections migrated by M-Apache.*

We run two load experiments on base Apache and on M-Apache to show that SC can improve availability of the service under load while not affecting server performance. The load is generated by httperf [15] clients running on four 300 MHz Pentium II PCs connected over 100 Mb/s Ethernet. The servers are 550 MHz, 1 GB RAM Pentium II PCs. We use a real trace collected on the Rutgers DCS web server over 18,370 files with average file size 27.3 KB and average reply size 19 KB. Each run of the trace lasts 300 s. A request that has not completed within 20 s is considered failed. The performance metric is the number of successful replies/s.

The first experiment shows how M-Apache exploits SC migration to improve overall server throughput. In this experiment, M-Apache migrates transfers that have not completed within 10 s. Figure 13 plots the aggregate client-perceived throughput for base Apache and M-Apache. Vertical bars show the number of connections migrated by M-Apache. We see that at small loads both base Apache and M-Apache perform identically and, as expected, M-Apache does very few migrations. However, while base Apache reaches saturation at 600 requests/s and "chokes" under increasing load, M-Apache continues to sustain good throughput well beyond this point by migrating ongoing transfers off of their origin server. When M-Apache reaches saturation at 800 requests/s, it can sustain twice the throughput of base Apache by migrating just about 10,000 of the 400,000 requests in the run. The average state size of a migrated SC was 9,386 bytes.

To understand this result, we compare the distribution of file sizes over all requests made to a server against the file size distribution on the migrated connections. Figure 14 plots the CDF of the file sizes over all requests, and over migrated requests. While most requested files are small (90th percentile less than 128 KB), file sizes for migrated requests are much larger (90th percentile larger than 384 KB). This shows that migrating requests for large files allowed the server to service more requests for smaller files and thus sustain a higher throughput.

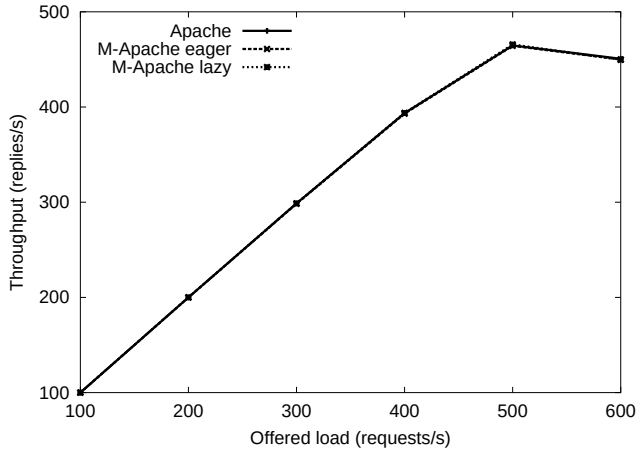

Figure 15: *Apache and M-Apache web server throughput with the Rutgers trace.*

Finally, note that the system was able to sustain up to 10,000 migrations over a period of 300 s under heavy load, showing its ability to handle high migration rates under load without losing performance.

The second experiment evaluates the run-time and memory overhead of using SC in M-Apache with the two variants of `export_state` primitive (eager and lazy). We compare base Apache against M-Apache under the same static load (i.e., without migrating transfers in M-Apache). The M-Apache snapshot amounts to a fixed part of the request of 18 bytes, plus, for our trace, a constant 24 bytes for the anonymized file name. Internally, a per-client SC uses an amount of system memory equal to the snapshot size for eager M-Apache, and a worst case of 5 memory pages for lazy M-Apache.

Figure 15 plots the reply throughput with increasing request rate until saturation, the perfect overlap of the three curves showing that M-Apache servers have performance similar to Apache regardless of load. The system memory overhead is negligible for eager, and a maximum of 1.44 MB for lazy M-Apache. We conclude that the use of SC to support migration in M-Apache has practically no impact on server or system performance.

Transactional DB Server. We have integrated migration support in PostgreSQL [18], an open-source transactional database system, and built a sample web-interfaced PostgreSQL application that uses SC and runs as a CGI script in our M-Apache web server. The resulting system allows a client to start a sequence of transactions with one front-end and transparently continue its execution on other front-ends if necessary, while preserving ACID semantics and deterministic execution. The design of Migratory PostgreSQL is described as a detailed case study in [25].

Audio Streaming Server. We have incorporated SC in Icecast [11], an open-source Internet audio streaming server, enabling client media player connections to transparently migrate between different servers, without interruption or distortion of the received stream. The SC API adds 350 lines to the base server code. In Icecast, a per-client state snapshot depends on the type of client media player, but can be reduced to a core of about 50 bytes in size. Its critical components are a stream identifier and one integer recording the client's position in the stream. The server exports a state snapshot after every 8 KB sent to a client.

8 Future Work

The SC idea opens interesting avenues of research in fine-grained OS support for fault-tolerant, self-healing and restartable systems and services. We plan to develop and apply it to these areas in our future research.

SC can be augmented with support for fault tolerance by making the volatile SC-encapsulated state persistent across server crashes. Two solutions are possible: *(i)* use some form of stable storage to store the SC state on the server side, or *(ii)* store the SC state with the client. In the first case, due to the highly dynamic nature of the SC state, the stable storage support has to be efficiently implemented. This is possible for example if the servers are connected (in a cluster) through a memory-mapped communication architecture like VIA [29]. This idea has been applied succesfully in previous work [32], where memory-mapped communication was used for fast failover of nodes in a cluster by establishing efficient checkpoints in the memory of other nodes.

A mechanism similar to SC can be developed and used in the area of system restartability for recovery-oriented computing (ROC) [16]. Restarting a system may be necessary, for example, because its internal state has been corrupted due to software bugs, environmental factors, etc. OS mechanisms may be used to preserve some portion of system and application state that can be salvaged, like that associated with serviced clients. A server application can later restart from this state and rebuild missing components based on internal semantics, external (logged) input, etc.

The SC idea can be further extended and viewed as the OS mechanism that enables transfer of a portion of "good" per-client state from one server (or multiple servers) to other server(s), if the rest of the state is compromised. The SC can be used as the abstraction of state distributed "horizontally" across multiple application processes and servers. In addition, SC can be extended to "vertically" extract and manipulate state from various OS layers that need to be replaced and/or restarted. An SC can therefore be used to (temporarily) transfer the clients to other servers while the system is being restarted.

We also intend to explore how SC can be used in IP-based storage networking for providing transparent fault tolerance and load balancing. Newly emerging industry standards for the transport of SCSI storage protocols over IP, like iSCSI [21], use TCP connections from a client to a "target" host that controls the storage device. A device server receives commands, relays them to the device, and returns results to the client. Such a system presents an interesing

case for SC, as client connections are long-lived, have unpredictable load, and are sensitive to host unavailability or device failure. SC can provide an immediate mechanism for offloading client iSCSI connections to alternate storage device servers, for example in case of an unexpected load surge or to hot-swap a host system. In systems with replicated storage, SC can provide transparent failover in case of device failures. We plan to study the possibility of using SC in the iSCSI protocol.

9 Related Work

SC is closely related to work in process migration, fault-tolerant operating systems, and in providing high availability for Internet services through protocol support.

Process migration [14, 30, 4, 9] is a heavy-weight, generic mechanism that enables seamless execution of an application process at multiple nodes during its lifetime, targeting load sharing and balancing in clusters. SC differs from classical process migration in that it trades off transparency for finer migration granularity, by application-level control of migrated state. While the goal of process migration is to transparently move and restore *whole execution contexts*, SC require the application to cooperate in defining fine-grained specific state distributed across multiple processes to be moved and restored in *different execution contexts*. SC addresses the problem of transparently restoring the state of open communication channels of an application when the execution site changes. Unlike most process migration schemes, SC does not freeze execution, and does not require inter-process or client-server synchronization. This comes at the expense of potential re-execution after a migration. Most existing systems supporting process migration are custom operating systems designed from scratch with built-in migration support [30, 4, 9]. In contrast, we provide a fairly straightforward implementation of SC in a general-purpose OS.

The idea of using logs at the OS level for deterministic execution replay can be traced back to early fault-tolerant operating systems like NonStop [5] and Auros [6, 7]. In the NonStop kernel, a message logging scheme was used to provide single-failure fault tolerance with primary-backup process pairs. This was probably the first system to use logging on inter-process communication channels to ensure deterministic replay at the OS level during fault recovery. SC uses a similar technique to synchronize the application and communication state, thereby ensuring deterministic behavior after a migration.

The SC synchronization scheme is similar to log-based rollback recovery techniques used in fault-tolerant distributed systems [10]. Unlike these, an SC does not use full checkpoints of process state, and limits the rollback and replay to restore only a small part of process state, specific to exactly one client.

Providing high availability for Internet services through protocol support has been approached in several ways: using connection migration in application-specific solutions [22, 31], fault tolerance for TCP [1], and new transport protocols [24]. None of these approaches provides the OS support required for migrating and resuming complex, multi-process service instances.

A scheme that enables HTTP connection endpoints to migrate within a pool of support servers is described in [22]. Migration is supported by broadcasting per-connection HTTP and TCP state within the server pool. The scheme 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 achieves transparency, it creates a strong dependency on the content of the application data stream and on HTTP specifics. Connection migration is limited to HTTP, for which it can only migrate static transfers. In contrast, our SC abstraction provides support for fine-grained connection migration, through a generic mechanism that can be used with any application, and which is not limited to single-process servers. With SC, a server application must change to assist migration. However, no knowledge of application specifics is required at the OS/protocol level for resuming the service after migration.

In [13], a technique for fault-resilience in a cluster-based HTTP server is described using a front-end dispatcher to monitor client connections serviced by back-end nodes. In case of failure, the dispatcher restores the serviced connections on another node using a scheme similar to [22]. The dispatcher is involved in correct CGI execution by caching dynamic content. The scheme is application-specific, limited to clusters, and makes the dispatcher a single point of failure and a potential bottleneck. In contrast, SC is a generic solution, does not use centralized control, can support multiple processes and works over wide area.

FT-TCP [1] provides masked recovery of a crashed server 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 client endpoints from the failure. The scheme uses full process context checkpoints to recover from a crash and works only for single process servers. Compared to FT-TCP, SC allows dynamic and fine-grained connection migration of client sessions with state distributed over multiple communicating processes.

TCP connection handoff is used in [3] for load balancing in clustered HTTP servers by distributing incoming client requests from a front-end host to back-end server nodes. Migration is limited to the initial connect. Multiple handoffs of persistent HTTP/1.1 connections at request granularity are mentioned, but no design or implementation are described. In contrast, SC allows dynamic migration between multi-process servers that can be distributed across a WAN. SC can be used in load-balancing schemes where load may be monitored at a finer granularity than a fixed, application-specific unit. For HTTP servers, SC supports

migration with dynamic content execution and persistent connections.

10 Conclusions

In this paper, we have introduced service continuations (SC), an OS mechanism that supports seamless dynamic migration of Internet service sessions between multi-process cooperating servers. Service continuations provide a server application with a simple and easy to use abstraction to migrate the service state, along with the serviced connection, to another server, at any point during service.

We have implemented SC in FreeBSD and present results of an experimental evaluation which shows that SC can efficiently provide support for dynamic migration of client sessions. We have succesfully used SC in three real applications: the Apache web server, the PostgreSQL transactional database server, and the Icecast streaming server.

A software distribution of SC/M-TCP in FreeBSD along with applications will be available soon. More details about the project can be found at our site: `http://discolab.rutgers.edu/sc`.

References

[1] L. Alvisi, T. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping Server-Side TCP to Mask Connection Failures. In *Proc. IEEE INFOCOMM '01*, Apr. 2001.

[2] Apache HTTP Server. http://httpd.apache.org.

[3] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient Support for P-HTTP in Cluster-Based Web Servers. In *Proc. USENIX '99*, 1999.

[4] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[5] J. F. Bartlett. A NonStop Kernel. In *Proc. 8th Symp. on Operating Systems Principles (SOSP)*, 1981.

[6] A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault Tolerance. In *Proc. 9th Symp. on Operating Systems Principles (SOSP)*, 1983.

[7] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance under UNIX. *ACM Transactions on Computer Systems (TOCS)*, 7(1):1–24, 1989.

[8] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end WAN Service Availability. In *Proc. 3rd USENIX Symp. on Internet Technologies and Systems (USITS)*, Mar. 2001.

[9] F. Douglis and J. K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[10] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[11] Icecast Streaming Server. http://www.icecast.org.

[12] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental Study of Internet Stability and Backbone Failures. In *Proc. 29th Symp. on Fault-Tolerant Computing (FTCS)*, June 1999.

[13] M.-Y. Luo and C.-S. Yang. Constructing Zero-Loss Web-Services. In *Proc. 20th IEEE Intl. Conf. on Computer Communications (INFOCOM)*, June 2001.

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

[15] D. Mosberger and T. Jin. httperf – A Tool for Measuring Web Server Performance, 1998.

[16] D. Patterson et al. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, Mar. 2002.

[17] J. Postel. RFC 793: Transmission Control Protocol, Sept. 1981.

[18] PostgreSQL. http://www.postgresql.org.

[19] Y. Rekhter and T. Li. RFC 1771: A Border Gateway Protocol 4 (BGP-4), Mar. 1995.

[20] Service Availability Forum. http://www.saforum.org.

[21] J. Satran et al. iSCSI, IETF Draft, Sept. 2002.

[22] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-Grained Failover Using Connection Migration. In *Proc. 3rd USENIX Symp. on Internet Technologies and Systems (USITS)*, Mar. 2001.

[23] K. Srinivasan. M-TCP: Transport Layer Support for Highly Available Network Services. Technical Report DCS-TR-459, Rutgers University, Oct. 2001.

[24] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzberger, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. RFC 2960: Stream Control Transport Protocol, 2000.

[25] F. Sultan et al. Migratory TCP: Highly Available Internet Services Using Connection Migration. Technical Report DCS-TR-462, Rutgers University, Dec. 2001.

[26] F. Sultan, K. Srinivasan, and L. Iftode. Transport Layer Support for Highly-Available Network Services. In *Proc. HotOS-VIII*, May 2001. Extended version: Technical Report DCS-TR-429, Rutgers University.

[27] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Connection Migration for Service Continuity in the Internet. In *Proc. ICDCS 2002*, July 2002.

[28] Test TCP (TTCP). ftp://ftp.arl.mil/pub/ttcp.

[29] The Virtual Interface Developer Forum. http://http://www.vidf.org.

[30] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proc. 9th Symp. on Operating Systems Principles (SOSP)*, pages 49–70, 1983.

[31] C.-S. Yang and M.-Y. Luo. Realizing Fault Resilience in Web-Server Cluster. In *Proc. SuperComputing 2000*, Nov. 2000.

[32] Y. Zhou, P. M. Chen, and K. Li. Fast Cluster Failover using Virtual Memory-mapped Communication. In *Proc. 13th International Conference on Supercomputing*, pages 373–382. ACM Press, 1999.