Migratory TCP: Highly Available Internet Services Using Connection Migration

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

Abstract

We evaluate the feasibility of using Migratory TCP (M-TCP), a reliable connection-oriented transport layer protocol that supports connection migration, for building highly available Internet services. M-TCP can transparently migrate the server endpoint of a live connection and assists server applications in resuming service on migrated connections. M-TCP provides a generic solution for the problem of service continuity and availability in the face of connectivity failures.

We have implemented M-TCP and present results of an experimental evaluation which shows it can efficiently provide support for highly available services. We illustrate the use of M-TCP in two applications. The first is a synthetic generic media streaming server. We show that, when the performance of the current server degrades, M-TCP can sustain throughput close to the average server behavior by migrating connections to better servers. The second application is a transactional database server in which we have integrated support for migrating client connections. Using our system, a database front-end can continue the execution of a series of transactions submitted by a remote client in a session started with another front-end. The system allows a session to survive adverse conditions by connection migration, while ensuring that ACID semantics are preserved and that the execution is deterministic across migration.

Keywords: reliable transport protocols, connection migration, Internet services, TCP, availability, fault tolerance

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 and applications requiring long-term connectivity are being developed and 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 [22], the standard Internet connection-oriented reliable 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 in between: congestion or failure in the network, server overloaded, failed or under DoS attack. As a result, 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.

The static service-server binding enforced by TCP limits its ability to provide service availability 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).

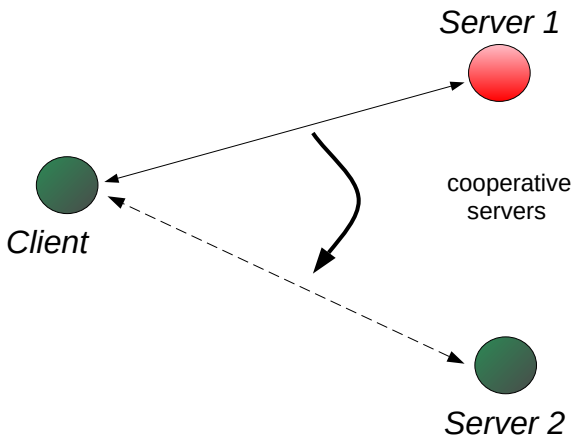

Figure 1: The cooperative service model

Retransmission occurs regardless of whether failure to deliver the packet is caused by an unreachable, down or unresponsive server, and cannot exploit the existence of an alternate server. In addition, loss of a packet triggers TCP's response to congestion using two well-known mechanisms: fast retransmit and slow start [15]. Research efforts have focused on improving these mechanisms [8, 12, 14, 19, 21, 29, 30], without altering their underlying assumption, i.e., that packet loss is caused by and is an indication of congestion in the core network. These observations point to the fact that TCP is overly limiting, as the end user may be more concerned with availability and/or quality of the service rather than with the exact identity of the server.

Server replication has been used as a solution to increase service availability by allowing clients to use any one out of potentially many equivalent servers, assuming that server identity is not important as long as a client can receive service from one of them. However, simple replication with TCP does not address the problem of *service continuity* after the connection is established. Subsequent network failure or congestion may render the service unavailable to the end user. Studies that quantify the effects of network stability and route availability [17, 9] demonstrate that connectivity failures 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 that may lead to disconnections between a typical client and a typical server (on the order of 15 minutes per day, according to [9]).

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 if the current server cannot sustain the service. This idea reflects a more powerful *cooperative service model* (Figure 1), in which a pool of hosts, 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, is carried over a dedicated, highly-reliable network, different from the one over which clients access the service. From a client's point of view, at any point during the lifetime of its service session, the remote endpoint of its connection may transparently migrate between servers. The only requirement imposed on the client is to renounce control over the identity of its server. In exchange, the service provider is responsible for accommodating the client on a new server in the event that changing the server endpoint becomes necessary.

To enable the cooperative service model and achieve high service availability, we have proposed Migratory TCP (M-TCP) [28, 26], a reliable byte-stream, connection-oriented transport layer protocol that supports live connection migration. When running M-TCP, hosts in a pool of similar servers can accept migrating connections and continue service on them. The server application must use a minimal protocol API to establish a fine-grained checkpoint of per-connection state, export it to the protocol, and resume the service from it on the destination host to which the connection has migrated. To migrate the server endpoint of a live connection, M-TCP transfers the last connection state checkpoint, along with protocol specific state, to the destination host. The protocol ensures that the destination server resumes the service while preserving

the exactly-once delivery semantics across migration, without freezing or otherwise disrupting the traffic on the connection. Although fine-grained connection migration solutions have been proposed for HTTP [24, 31] by exploiting details of the protocol, to our best knowledge M-TCP is the first solution that provides generic migration support through a TCP-compatible transport protocol.

The goal of this paper is to demonstrate that M-TCP is a viable solution in building highly available services over the Internet, and it can successfully overcome problems that confront other protocols. We have implemented M-TCP in FreeBSD and conducted experiments with synthetic streaming applications to show its feasibility in supporting continuous streaming service over reliable connections. We show that, by using adequate migration policies, M-TCP can sustain throughput close to the average server behavior by migrating connections to better servers.

We also show that M-TCP can provide availability for a real-world distributed application - remote access to a transactional database system over the Internet. We present a detailed case study in design and implementation of migration support for high availability with M-TCP. To validate the design, we have integrated migration support in PostgreSQL, an open-source database system, and built a sample web-interfaced application. The resulting system allows a client to start a sequence of transactions with one front-end and continue the execution on other front-ends if necessary.

The remainder of the paper is structured as follows. In Section 2, we review related work in the area of using transport layer protocol support for high availability. In Section 3, we present an overview of the M-TCP protocol. Section 4 discusses the issues related to applications built over M-TCP. Section 5 presents experimental results supporting building media streaming services over M-TCP. Section 6 makes a case study in the design and implementation of migration support for transactional database services over Internet. Section 7 discusses limitations of the current M-TCP implementation. Finally, Section 8 concludes the paper.

2 Related Work

High availability of Internet services through transport layer support has been approached in several ways: fault tolerance for TCP [4], new protocols like SCTP [27], and using connection migration in application-specific solutions [24, 31]. Connection handoff protocols have been used in several mobility extensions to TCP [6, 7, 25] but their relevance to highly available services is marginal.

A fail-over scheme that enables HTTP connection endpoints to migrate within a pool of support servers is described in [24]. Migration takes place in response to a failure and is supported by broadcasting of per-connection and soft TCP state within the pool. The scheme adds a HTTP-aware module at the transport layer that extracts information from the application data stream and uses it for connection resumption. While the design leads to no change in the server application, it is application-specific, as it relies on knowledge of the upper layer protocol. M-TCP also provides support for fine-grained connection migration, but through a generic mechanism that can be used with any application. With M-TCP, a server application must change to assist the transport protocol. However, no knowledge of application specifics is required at the protocol level for resuming the service session after migration.

The Stream Control Transmission Protocol (SCTP) [27] is a recently proposed transport layer protocol targeting streaming applications. It provides sequenced, reliable delivery of datagrams within multiplexed streams between two endpoints. SCTP uses endpoint multi-homing to exploit path redundancy between two endpoints to tolerate network failures. With SCTP, existence of distinct paths between the *same* endpoints, which may not always be possible, is critical to availability. M-TCP can also exploit existence of alternate internetwork paths for high availability, but to different endpoints, which makes it more powerful and flexible.

FT-TCP [4] is a scheme for transparent (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 remote endpoints from the failure. The scheme is coarse-grained, using a full process context to recover from a crash. In contrast, M-TCP can use fine-grained connection migration in response to other types of adverse events beside a crash. M-TCP can be combined with FT-TCP to support

highly available services through fault-tolerance at the process level, while allowing recovery of individual connections, possibly on different nodes.

MSOCKS [20] is a proxy based scheme providing continuity in service sessions to mobile clients equipped with multiple interfaces. TCP connections are spliced at the proxy to enable a session to use different interfaces. Using this scheme the client may get better connectivity on the first hop to the server (via the proxy), but it is still bound to the same server and may suffer from connectivity failures inside the proxy-server network.

Mobile TCP solutions maintain TCP connections in the presence of host mobility. They either rely on directly using the full connection state maintained at the physical endpoints [6, 7] or on restarting a previously established connection after a change of IP address by reusing the state of the old connection with proper authentication [25]. As a marginal benefit, a mobile TCP might improve client-perceived performance in face of network failures by using potential alternate routes to a server from a new location. Unlike M-TCP, none of these schemes consider the task of migrating connection endpoints between physically distinct machines.

TCP handoff is used in [5] in clustered HTTP servers for load balancing by distributing incoming client requests from a front-end host to back-end server nodes. The system employs a limited, single handoff scheme, in which a connection endpoint can only migrate during the connection setup phase. Multiple handoffs of persistent HTTP/1.1 connections are mentioned only as an alternative, but no design or implementation are described. Even in the multiple handoff case, the granularity of migration of live connections is application-dependent: a connection can migrate only after fully servicing a HTTP request. In contrast, M-TCP allows dynamic connection migration at any point in the byte stream, between servers distributed across a WAN. M-TCP can be used in a load-balancing scheme where the load could be monitored at a finer granularity than an application-specific unit.

In [31] a technique for fault-resilience in a cluster-based HTTP server is described using a front-end distributor to monitor the state of client connections serviced by back-end nodes. In case of node failure, the distributor restores the serviced connections on another node. The scheme is application-specific, limited to clusters, and makes the distributor a single point of failure and a potential bottleneck. In contrast, M-TCP is application-independent, does not use centralized control and works over wide area.

3 Overview of the M-TCP Protocol

Migratory TCP (M-TCP) [28, 26], is a reliable connection-oriented transport layer protocol that supports efficient dynamic connection migration of live connections. In addition, the protocol enables an application to resume service by transferring an application-controlled amount of specific state with the migrated connection.

3.1 Connection Migration and Cooperative Service Model

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 supporting state. Migration is dynamic, in the sense that it may occur multiple times, at any point during the lifetime of the connection, without disrupting the ongoing traffic.

3.1.1 Per-Connection State

To migrate a live connection, M-TCP transfers associated state (protocol and application-specific) from the origin (old) to the destination (new) server. Essential to M-TCP is the assumption that the state of the

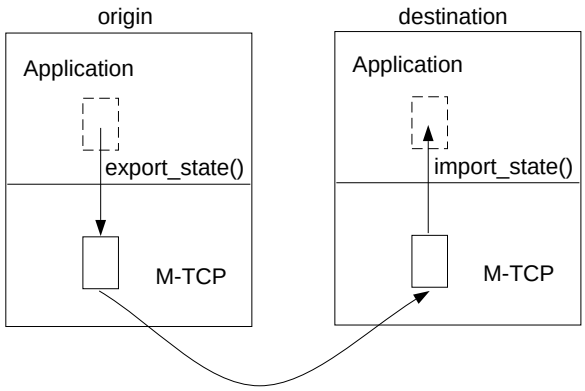

Figure 2: *Manipulation of a state snapshot as part of the service contract.*

server application can be logically split among connections, so that there exists a well-defined, *fine-grained* state associated with each connection. Any other non-specific state needed to resume service on a migrated connection is deemed accessible at the new server.

Transfer of protocol connection state (e.g., sequence numbers information, buffered unacknowledged segments, etc.) reconciles the protocol layer of the new server with that of the client. In general, the state reached by a server process while servicing a client cannot be inferred from the protocol state of the connection it uses. To support continuity of stateful services, M-TCP also transfers application-specific state. This application-specific state defines a *restart point* in the service session.

3.1.2 Migration API and Service Guarantees of M-TCP

M-TCP provides a minimal API that can be used by a server application to enable endpoint connection migration: for a given connection, the application would export/import a *state snapshot* of the associated application state to/from the protocol. The state snapshot is opaque to the protocol but must completely describe the point the application has reached in an ongoing session, so that it can be used as a restart (reference) point after a connection migration.

The two main primitives of our proposed API are: `export_state(connid, state_snap, size)` and `import_state(connid, state_snap, size)`, where `state_snap` is an application memory buffer holding the state snapshot of size `size` for connection `connid`.

The M-TCP service interface can be best described as a *contract* between the application process and the transport protocol. According to this contract, the application must execute the following actions:

- *export* the per-connection application state (the state snapshot) at the old server, when it is consistent with data sent/received on the connection;
- *import* the last state snapshot at the new server after migration and resume service to client.

In exchange, the protocol:

- *transfers* the per-connection state to the destination server;
- *synchronizes* the per-connection application state with the protocol state.

Figure 2 shows the export/import steps by the server application and the transfer of state by the protocol between the two server hosts. If the application follows the terms of the contract, the protocol guarantees dynamic connection migration of the local endpoint without loss of reliability in the data stream. The migration API decouples the server application from the actual migration time by enabling asynchronous connection migration, makes the scheme light-weight and yields good performance.

3.2 Migration Mechanism

The mechanism for connection migration in M-TCP reincarnates the migrating endpoint of the connection at the destination server and also establishes a *restart point* for the server application. To achieve this, the migration mechanism *transfers* the state associated with the connection (protocol state and application snapshot) from the old to the new server. Depending on the implementation, the transfer of state can be either *(i)* lazy (on-demand), i.e., it occurs at the time migration is initiated, or *(ii)* eager, i.e., it occurs in anticipation of migration, e.g., when a new snapshot becomes available.

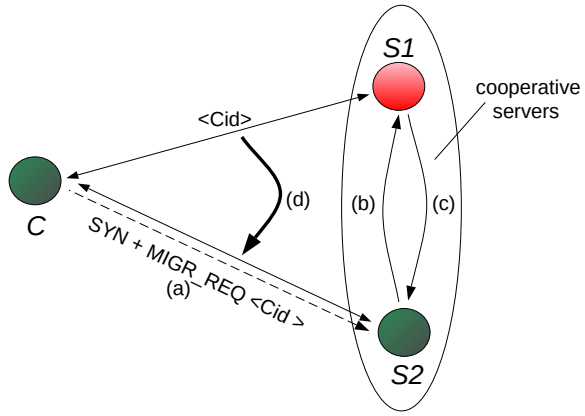

Figure 3: *Migration mechanism in M-TCP. Connection C_{id}, initially established by client C with server S_1, migrates to alternate server S_2.*

Figure 3 describes the sequence of steps in a migration. In the beginning, the client contacts the service through a connection C_{id} to a preferred 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 initiates migration of C_{id} (according to some policy) by opening a new connection to an alternate server S_2, sending the migration certificate in a special option (Figure 3 (a)). To reincarnate C_{id} at S_2, M-TCP transfers associated state from S_1. Figure 3 shows the lazy 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, at which point C_{id} is switched over to the new connection (d). All these steps are transparent to the client application.

The in-kernel protocol data buffering and the active send/receive communication model cause the protocol state of a connection at a given moment in time to go out of sync with respect to the state reached by the application as a result of sending/receiving data on the connection. To solve the synchronization problem for the two components of the connection state, M-TCP uses a limited form of *log-based recovery* [11] to restore the state of the application at the new server. The mechanism works as follows:

- Upon accepting a migrating connection, the application running at the new node *(i)* imports the last state snapshot from the protocol, and *(ii)* initializes the local session state using the imported state snapshot and resumes execution.
- After resuming execution, the application may *replay* execution already done at the old node since the snapshot. This includes receiving data that was received at the old node, and sending data that was already sent.
- To support the replay of received data, the protocol *logs* and transfers from the old node data it has received and acknowledged since the last snapshot. The data sent before the last snapshot and not acknowledged is also transferred [1].

[1] The key assumption for a log-based recovery scheme is that the application follows a *deterministic process execution model*. We believe that this assumption is not overly restrictive.

The most recent application state snapshot represents the *restart point* of the service session at the new server and, along with the protocol state of the connection, enables the new server to do the following: *(i)* restart servicing the client session from the state snapshot on, *(ii)* replay data received at the old server which cannot be supplied (retransmitted) by the client-side (because it has already been acknowledged from the old server, and *(iii)* re-send data sent from the old server before the last snapshot (i.e., which cannot be re-generated by execution replay at the new node), and which had yet to be acknowledged when migration occurred.

The above three guarantees, coupled with a deterministic execution model, ensure that the new server can resume service to the client consistently after migration.

3.3 Implementation Features

In [26] we describe an implementation of M-TCP with lazy (on-demand) state transfer between the origin and destination server. In this particular implementation, state associated with a connection is transferred only in response to initiation of migration. Other mechanisms, like eager state transfer or state replication at the client, are also possible. The implementation supports efficient state transfer between cooperating servers through a dedicated control network which allows M-TCP control traffic to be decoupled from client traffic (requests/replies).

M-TCP is compatible and inter-operates with the existing TCP/IP stack. We implemented M-TCP as an extension to TCP, and the M-TCP control packets 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 negociate their capabilities in terms of support for migration. If either of the parties does not run M-TCP then the connection defaults to a regular TCP connection. If both the client and the initial server are migration-enabled, the client side of the connection obtains a list of cooperating peers, to be used in case a migration becomes necessary.

From the point of view of application awareness, migration is completely transparent to the client application. For the server application, the fact that a connection has migrated simply translates into an `EMIGRATED` error that the server may get when attempting its next operation on the migrated endpoint. The protocol is optimized to allow maximum possible overlap of migration-related operations with data delivery during migration at both endpoints, so that migration has minimum impact on application execution. For example, a client application may continue to receive and use data from the old server even after the client protocol initiates connection migration to a new server. Also, the client can continue to write on its endpoint of the connection while the migration is in progress. The actual switch to the new server is delayed until the state of the migrating endpoint has been properly reinstated at the new server and data transfer can resume in both directions.

3.4 Migration Policies

M-TCP provides only the mechanism to support migration of live connections. In [28] we proposed a migration architecture which decouples the 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, connection migration can be used: *(i)* on 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)* on the server side, for example to implement a load balancing scheme by shedding load from existing connections to other less loaded servers, or an internal policy on how the content should be distributed among groups of clients. The experimental evaluation in Section 5 uses a sample policy designed for experimental purposes which can also be applied in practice. Definition and evaluation of migration policies is beyond the scope of this paper.

The choice of a policy may be largely application-dependent. At a minimum, a migration policy should define a metric, along with trigger events for migration based on that metric. Several simple policies can be defined starting from requirements of various classes of applications. For example: *(i)* For soft real-time applications

which are delay-sensitive, delay or jitter in receiving the next in-sequence segment could be an appropriate metric. *(ii)* For critical applications, like those that involve bank transactions, the user-perceived response time is critical; a proper metric in this case could be the response time or the RTT. *(iii)* For throughput-sensitive applications, the estimated inbound data rate could be used as a metric; an appreciable decrease in throughput over a period of time would warrant a migration.

4 Use of M-TCP in Server Applications

4.1 Impact of Migration API

With M-TCP, the server application must obey a certain programming discipline by calling primitives for exporting/importing to/from the protocol the application level state associated with a potentially migrating connection. However, the client application is not required to change, which allows existing client applications to benefit right away from the high availability support brought by M-TCP.

Also, the application may have to exploit certain performance tradeoffs. One possible tradeoff is between the size of the application state snapshot and the size of the protocol state. Delaying a snapshot for too long in order to optimize its size may increase the amount of data to be logged by the protocol. Another tradeoff is between the size or frequency of the state snapshot and the amount of work redone at the new server after a migration. In some cases, increasing the snapshot size may capture more computation and might result in less work being redone after restarting at the new server. However, larger snapshots could increase the runtime overhead and the time spent in migration.

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

Figure 4 illustrates the use of the M-TCP migration API with the pseudocode of a sample HTTP server serving a static request (which, for simplicity, contains only the file name). The state snapshot records the file name and the offset reached by the server while sending it. The origin server exports the snapshot with an `export_state()` call, after sending a block of data to the client. The destination server accepts a migrated connection, retrieves the snapshot with an `import_state()` call, and uses it to open the file and reposition its offset to correctly resume transfer. A server discriminates between a migrating and a new connection using a boolean value returned by the `import_state()` call. Note that the same code is run at all servers.

4.2 Applications of M-TCP

Under the cooperative service model, M-TCP 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 or DoS attack, network failure or congestion on a given path, etc. Migrating the connection to another server in case the service becomes unavailable ensures that sustained service is delivered to the client.

To efficiently benefit from M-TCP, an application must be willing to incur the cost of migration. In any case, migration should amortize its cost 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 M-TCP:

- *Applications that use long-lived reliable connections.* Examples are multimedia streaming services, applications in the Internet core that use TCP communication over large spans of time, etc.

  Recent research in support for multimedia communication tends to refute the commonly held belief that unreliable transport is the best choice for network multimedia applications. For example, [16] strongly argues for streaming over the reliable connection-oriented TCP as a viable alternative to connectionless,

```
struct {
   char fname[1024]; // file name
   int off;          // file offset
} state_snap

while (accept(conn)) {
  if (import_state(conn, &state_snap, sizeof(state_snap))) {
    fd = open(state_snap.fname)
    set_file_offset(fd, state_snap.off)
  } else {
    receive(conn, &state_snap.fname)
    fd = open(state_snap.fname)
  }

  while (not eof(fd)) {
    read(fd, buff)
    send(conn, buff)

    state_snap.off = get_file_offset(fd)
    export_state(conn, &state_snap, sizeof(state_snap))
  }
}
```

Figure 4: Sample HTTP server with connection migration support

unreliable transport. This view is also supported by ongoing work in the Internet community on connection-oriented protocols like SCTP [27], specifically designed for streaming multimedia, and which include all classical mechanisms for reliability and congestion control devised in TCP.

We believe that M-TCP can not only provide increased availability of streaming services over Internet by connection migration, but it can also be a useful addition to the work on QoS assurance for streaming applications. A topic of future research would be to devise proper migration policies based on well-known QoS metrics for such applications.

- *Critical applications.* Characteristic to this class of services is that users expect both correctness and good response time from the service. Examples are Internet banking and e-commerce. We believe that use of M-TCP can add value to such applications. In Section 6 we make a detailed case study of integrated system support for this class of applications. To prove usefulness of M-TCP in this context, we have integrated front-end migration support into a transactional database and thus enabled client connections to migrate between front-ends running on different nodes.

A notable example of how M-TCP can be used in a non-trivial application is the Border Gateway Protocol (BGP) [23] of the Internet core. BGP routers set up TCP connections between them to exchange both full routing tables and routing table updates. The connections must be up throughout the lifetime of the router. Taking a router down for maintenance or upgrade can seriously disrupt the protocol, leading to unacceptable recovery times [18]. An alternative is to use M-TCP to push the live connections along with their associated routing state to other machine(s) that can take over the routing task.

5 M-TCP Implementation and Evaluation

The goal of our evaluation is to validate the use of M-TCP to support streaming applications using two experiments. In the first one we use a microbenchmark application to determine the time taken to complete migration by the client-side protocol. This gives a measure of protocol responsiveness to migration triggers.

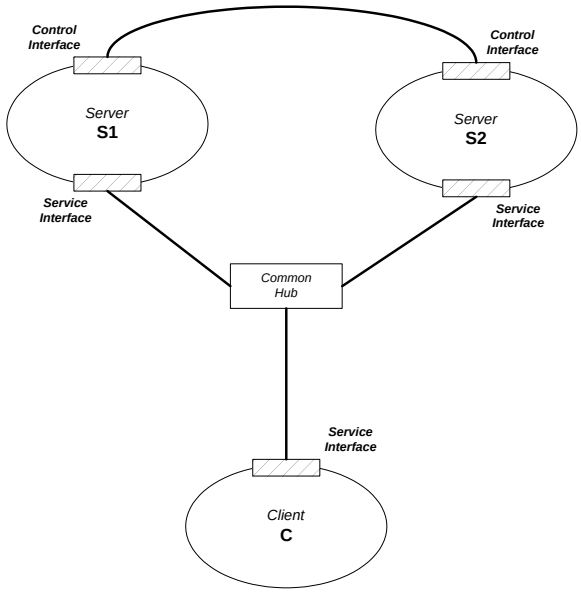

Figure 5: Experimental test-bed

The second experiment tests the use of a sample migration policy based on estimated inbound data rate to sustain good streaming throughput when server performance degrades.

Our experimental setup (Figure 5) consists of three Intel Celeron 400MHz PCs with 128 MB RAM connected by 100 Mb/s Ethernet through a dedicated hub, running FreeBSD 4.0 with M-TCP implemented as an extension to TCP/IP. All experiments ran cooperating servers on two of the nodes and a client on the third.

The server machines were each equipped with a second 100 Mb/s Ethernet interface, dedicated to transfer of connection state by M-TCP. The server applications service client requests on the main (service) interfaces, while M-TCP uses the secondary (control) interfaces to transfer state in support of migration. This setup reflects support by our M-TCP implementation of a real-world scenario in which cooperating servers can be interconnected by a fast and reliable dedicated control network to support client migration efficiently.

For the first experiment we used a migration-aware synthetic server application which did not perform any actual service. This eliminated variability in per-connection protocol state while we could vary the state snapshot size conveniently and measure the time needed to complete a migration. Measurements for a particular snapshot size were averaged over 200 runs, each run consisting of a single one-way connection migration of a client between servers. The graph in Figure 6 plots the time needed for migration completion, measured at the client-side M-TCP, versus the state snapshot size. The graph validates the intuition of a linear dependency between snapshot size and migration time. The startup value of 387 μs compares fairly well against an average RTT of 150 μs on our test network[2]. Note that the time measured in Figure 6 represents intervals between two *protocol* events, i.e., between the moment when the client-side protocol software initiates a migration to a new server and the moment when data exchange can start. As mentioned before, M-TCP heavily overlaps migration with delivering data from the old server, so the time to complete migration does not necessarily reflect a gap in communication for the client application.

The second experiment tested the sensitivity of the protocol when used with a rate-based migration policy in the presence of server performance degradation. We used a streaming application, where the server sends data at regular intervals, as a sequence of chunks of 1 KB. The length of the stream was limited to 256 KB. After sending a chunk, the server takes a snapshot recording the position it has reached in the stream. The experiment was conducted for server snapshot sizes of 2 KB, 10 KB and 16 KB. After sending 32 KB

[2]In the implementation under study, where state is transferred lazily between servers, the lower bound for migration completion time is 2 RTT.

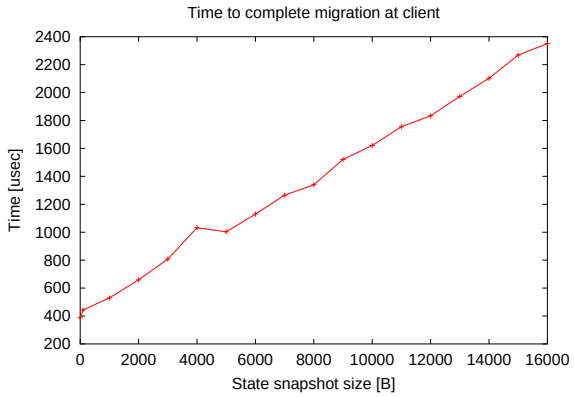

Figure 6: Time to complete a migration by client-side M-TCP

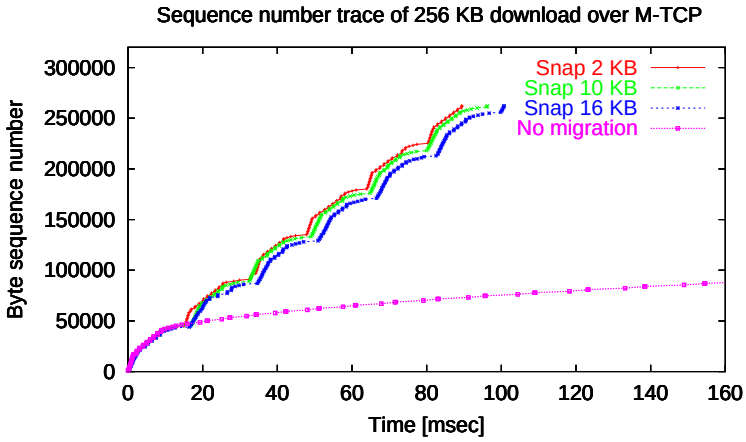

Figure 7: Traces of data received from a streaming service over a regular TCP connection, and over three migrating M-TCP connections with different server snapshot sizes. The server performance decays after sending 32 KB on any incoming connection.

on a connection, we simulate a degradation in server performance by gradually increasing delays between successive chunks sent. This behavior affects the throughput as perceived by the client.

We have implemented a simple policy module on the client side, that uses as metric the inbound data rate and triggers migration when the estimated rate drops under a threshold. We use a simple smoothed estimator S_rate for inbound data rate with a low-pass filter of the form $S\_rate = \alpha * M\_rate + (1 - \alpha) * S\_rate$, and a smoothing factor $\alpha = 1/8$, where M_rate is the measured rate sample. While this may not be a proper rate estimator for TCP traffic [10], it serves its purpose in this experiment. As the delays engineered at the server are gradual, the estimator gives a good measure of the effective rate. The policy module triggers a migration when the estimated rate drops by 25% from the maximum rate seen on the connection from the current server.

Figure 7 shows the trace of byte sequence numbers received by the client, up to the maximum of 256 KB, where we cut the stream. The graph shows four cases, one based on conventional TCP (no migration) and three with M-TCP with different per-connection state snapshot sizes. With M-TCP, migration takes place alternatively between the two servers, every time the current server becomes "too slow." Each discontinuity in the slope corresponds to a migration, after which data is received at the best rate from the new server. The

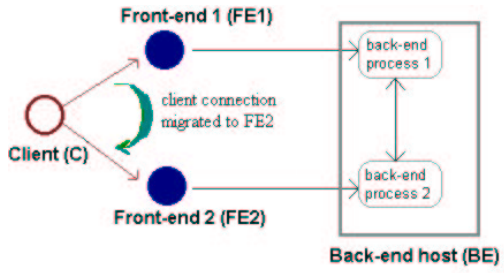

Figure 8: Connection migration used in a client-server database system architecture.

net result is that the effective rate at which the client receives data is fairly close to the average rate at which a server sends before its transmission rate drops dramatically. The graph also exhibits the cumulative effect of the time taken by each migration and of the server 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 M-TCP for sustained streaming service by connection migration.

6 Highly Available Internet Database Services Using M-TCP

In this section we present a proof-of-concept implementation of server-side migration for high availability of transactional client-server applications over WAN (Internet). The scenario where this might be useful is when a client connects to a transactional database server, starts a series of one or more transactions, and expects one or more replies with transaction results. Receiving a reply may become impossible or be delayed due to network failure or a sudden spike in server load. In this case the end user has to wait indefinitely for the outcome of the transaction, or to cancel it and restart.

The solution we propose is to use our migratory transport protocol to switch the live client connection over to another cooperative server that can continue execution of client's transactions and/or provide the results to the client. The server switch takes place transparently to the client application.

Migration support must be included in the front-end by using the M-TCP API. While this provides migration support for the *communication* side (as described in previous sections), support for migration may also be needed on the *database* side, to ensure that *(i)* ACID properties are preserved, and *(ii)* execution replay after migration is deterministic. The second requirement is important, as execution replay after migration at the destination server may depend on contents of the database that could have concurrently changed since the client started execution at the origin server. For the front-end and the back-end server, our current implementation forces the switch to occur at the granularity of a transaction, i.e., the largest unit of work that the back-end may have to re-do on behalf of a client is one transaction.

6.1 System Model

Figure 8 describes the system model for our study. The database side of the system largely reflects the architecture of the PostgreSQL [3] relational database system that we used in our implementation. PostgreSQL supports transactions and, with the exception of some implementation details (e.g., one back-end server process per front-end), it is general enough for the type of systems we target. The client C uses a reliable M-TCP connection to communicate with a front-end (FE) which provides the access interface to the back-end server (BE). The client sends a request to a front-end, the format and content of which are defined by the C-FE interface and by the specific application (for example, the request may be sent through HTTP and it may specify batched execution of a series of transactions on the database). The front-end receives requests from the client, translates them into database queries and submits the queries for

execution to the back-end. The back-end server manages the data repository, where, as shown in the figure, it may use a dedicated, per-FE process, to service front-end queries. The back-end executes the queries (potentially changing the database in the process) and returns results to the front-end, which (possibly after some processing) sends replies to the client.

This sample architecture is widely used in web-interfaced databases, where the role of the front-end is to shield the HTTP client from details of the back-end database: a FE handles the communication with the HTTP client, interpreting and translating its requests into native requests to the back-end, which only does the database work.

6.2 Migratory Front-Ends

The above architecture can be easily mapped onto the cooperative service model described in Section 1. The *cooperative service* provided to the client is that of interfacing it with a transactional database it needs to access. The *cooperating servers* providing this service are the front-ends running on distinct hosts. We assume a client is not interested in any particular server host (FE) providing answers to its requests, but is interested in obtaining the replies reliably, and within a reasonable time bound. Therefore, it is reasonable to migrate the FE_1 endpoint of the initial connection to satisfy a client's expectations about the service in terms of availability or response time.

The client-side protocol software can trigger a migration of the C-FE connection based on some pre-set policy, for example using as metric the number of retransmissions on the connection or the estimated RTT. Another front-end FE_2 capable of providing the service (i.e., interface to back-end) resumes execution of client's transactions from where FE_1 left off.

In this model, the FEs act as stateful servers in support of migratory clients. They take state snapshots to establish restart points using the M-TCP `export_state()` API call. At a minimum, they should record in the state snapshot the point reached in a sequence of transactions. Note that some state reflecting the ongoing queries on behalf of the migrating client is also necessarily maintained by the back-end process BE_1 that executes queries issued by FE_1 on behalf of C. As a result, migrating the client to FE_2 also requires reinstating at its corresponding BE_2 the back-end context in which FE_1's queries were executing before migration. This may involve, if needed, a state transfer between BE_1 and BE_2 (illustrated by the double pointed arrow in Figure 8) at the time FE_2 reconnects to the database to resume service to client after migration. While this is an implementation specific detail, it illustrates the kind of problems that may need to be addressed on the database side to support migration.

Note that migratory front-ends can also address availability problems of the FE-BE network. A slow or faulty link between FE_1 and BE would result in bad service perceived at the client side, and thus trigger migration to a front-end with better connectivity to the back-end. Also, note that the focus of our solution is not on back-end availability, which is assumed to be provided by the implementor of the database.

6.3 Migration Granularity

We define the *granularity of migration at application level* as the extent of application execution that *could* be re-executed after a restart at FE_2 from a state snapshot. In transactional applications there is a clear upper bound for migration granularity. The largest unit of work that can be safely re-issued to the database at FE_2 after a migration is one transaction. Any other larger unit (for example, a chain of several transactions) cannot be guaranteed to yield the same results at FE_2, due to potential side-effects of previous transactions in that unit that were issued and committed at FE_1. This would contradict the assumption of a deterministic execution model for the front-end application. This upper bound imposes a discipline on the FE, which must take snapshots at least every transaction, to avoid unwittingly re-doing transactions in case of migration. A related issue, equally subtle and important, is exactly *when* the state snapshots are to be taken. We will show how migration granularity imposes a discipline on the placement of state snapshots.

While the upper bound on migration granularity is strictly set by correctness constraints with respect to application execution, the lower limit, which defines the granularity at which migration can actually occur

within an executing transaction, is open to design decisions and application behavior. Choosing this limit involves a trade-off between the application runtime overhead incurred by the migration support and the amount of work lost (redone) as a result of migration. Allowing applications to use a finer granularity means little work needs to be redone at FE_2, but may incur unacceptable overhead in terms of frequency and size of state snapshots (needed to store intermediate query results). Coarser granularity means less overhead, but may incur re-doing more work after migration at FE_2. Note that the state snapshot size greatly depends on the application: if the application issues queries resulting in only a few bytes of data which need to be saved in the snapshot, then it is reasonable to trade the small size for a higher snapshot frequency and use fine-grained migration.

In our PostgreSQL implementation we had two choices for migration granularity: *(i)* query level, or *(ii)* transaction level. Note that the decision determines the kind of support the back-end must provide for migration, since we must be able to associate any state snapshot taken by a front-end with a well-defined state of its associated back-end context. As the decision is implementation dependent and involves back-end details, in the following discussion we concentrate on the details of our PostgreSQL implementation. Equally important, as we will show, the choice also determines a programming model for the front-end application.

6.3.1 Query-Level Granularity

In PostgreSQL, the smallest unit of work that a BE accepts from a FE is called a *query*. The query contains a valid SQL verb along with its parameters. The BE executes the query and returns a result in response. A transaction is a sequence of queries starting with a *begin* query and ending with a final query that can only be *commit* or *abort*. The usual ACID semantics [13] are guaranteed.

If migration were supported at query level, FE_2 could resume execution of a transaction starting from the last query issued at FE_1 when migration occurred. This would completely eliminate redundancy in re-issuing queries at FE_2 that were already done at FE_1. However, there are a few subtle issues with this approach:

1. Locks acquired by the transaction have to be held during migration. The trade-off between the time required for redoing the queries and the delay incurred by holding the locks across migration depends on the complexity of the queries and the lock contention in the back-end. Holding locks across migration may negatively impact the degree of concurrency in the back-end.

2. Results of queries executed early in the transaction might be used by later queries. Since past queries (before a snapshot) cannot be re-executed at FE_2, their results need to be logged in order to be preserved across migration. Recording results at a FE may require it to take state snapshots after every query is executed. This has the potential of generating huge state snapshots (for example, a `SELECT` on all rows of a table is a valid query that may generate large amounts of data). The alternative is to keep a log of past query results at the back-end, and discard the log when they are no longer needed. This requires an extra call in the database API and makes the application aware of the presence of this log. Also, depending on the application, the size of the log could be very large, which consumes resources at the back-end.

3. Supporting migration at the granularity of intermediate queries in a transaction is easy if the transactional database supports *savepoints* [13], which would allow partial rollbacks within a transaction, without affecting its overall outcome. In this case, a FE does not need to take snapshots after every query but instead would take a snapshot before a savepoint to which it may need to rollback. Upon resuming execution, FE_2 rollbacks to the previous savepoint, at an application-controlled granularity. This approach eliminates logging, but incurs the cost of re-executing several queries.

6.3.2 Transaction-Level Granularity

In this case, after migration, FE_2 resumes execution starting with a transaction following the most recently committed or aborted transaction at FE_1. The transaction currently executing at FE_1 (which has not

yet been committed or aborted) must be aborted by BE_1 when migration occurs. This is because the new incarnation of the transaction restarted at FE_2 should not be competing for resources with a similar transaction, now useless, running freely at FE_1. For correct resumption after a migration, a FE must take snapshots when is about to issue the final query (*commit* or *abort*) for a transaction. This is the point where all transaction's work has been done and its *outcome* is known.

Recall that migration may occur at any time and is transparent to FE_1, at least until it attempts an operation on the migrated endpoint (e.g., exporting a state snapshot or sending data to C), which returns an error. A snapshot before the final query in a transaction is needed to ensure that the transaction is not duplicated at FE_2 after having committed/aborted at FE_1. The snapshot must be exported *inside* the transaction, protected by the final query, to make it atomic with respect to migration.

Suppose instead that a snapshot is taken after the final query, for example a commit. It is then possible that migration occurs right after commit but before FE_1 is able to take a snapshot recording that the current transaction is about to end. The previous snapshot would then be used by FE_2 to restart with (and thus incorrectly duplicate) the *same* transaction that has just committed at FE_1.

On the contrary, a snapshot taken before commit records the fact that the current transaction has done all its work and is about to end. If migration occurs after snapshot but before commit, FE_2 can retrieve the outcome of the transaction upon reconnecting to the back-end and decide, according to application logic, how to proceed next based on it. Note that in this case there is a race between the transaction committing at FE_1 and the check for its outcome from FE_2. A failure of FE_1 before commit would be detected by BE_1 and automatically abort the transaction, so in this case FE_2 is guaranteed to get a result. However, if FE_1 cannot commit for some other reason (e.g., overload), this could delay processing at FE_2 indefinitely. Migrating again to another FE would not help since the real cause of the problem is the origin node FE_1. We solve this problem by aborting the transaction at BE_1 if it has not issued its final query after a timeout from the moment FE_2 checks for its outcome. Two important things to note about this forced abort. First, although it may kill a transaction that would have otherwise committed, it allows the client to make progress at FE_2. Recovery is left to the client, which can safely retry the transaction through FE_2. Second, recovering by re-doing the transaction at FE_2 would be wrong as it violates the deterministic re-execution starting from the last snapshot (taken before the final query of the transaction).

Forcing snapshots before the final query of a transaction (commit/abort) provides a clean programming model for migration at transaction granularity and solves the problem of a race between the final query and the asynchronous transparent migration of the client connection. We have two cases: *(i)* If a migration occurs during a transaction *before* exporting the snapshot, then the export would return error, which forces FE_1 to abort the current transaction and terminate. *(ii)* If migration occurs between the snapshot and the final commit/abort query, then FE_2 is guaranteed to obtain a definite outcome of the transaction.

Advantages of transaction-level granularity are the clear programming model and that locks need not be held during a migration. They come at the potential cost of redoing queries already executed at the origin front-end.

6.4 Implementation Details

We have integrated migration support in PostgreSQL 7.0.3 [3], an open source relational database system, which has an architecture similar to that depicted in Figure 8. A "postmaster" process runs on the back-end host handling all incoming front-end connections to the database. When a FE connects to it, the postmaster spawns a BE process which handles queries from FE over a TCP connection. Clients connect to a front-end over M-TCP connections (enabled for migration).

We chose to support migration at transaction granularity because of a cleaner implementation, given the absence of support for savepoints in PostgreSQL.

6.4.1 Changes to the Front-End

We extended the PostgreSQL front-end API to include two calls: `reconnectDB` and `gettxnID`. The first call queries the back-end for the transaction identifier (*tid*) of the transaction in progress. The second call allows FE_2 to recreate at BE_2 the context for resuming execution. It takes as argument the *tid* of the transaction in which the imported snapshot was taken at FE_1 and returns its outcome.

A FE must export (before a transaction's final query) a state snapshot including: the *tid* of the transaction in progress, the logical sequence number of the last transaction executed, and any results of the transaction that might be needed. A commit/abort does not return a result, so all the results of the transaction are available at this point and can be exported in the snapshot.

As discussed before, because migration is transparent to the origin front-end, FE_2 may race with a transaction still executing at FE_1. This race is detected in the back-end when FE_2 calls `reconnectDB`: if BE_1 is executing the transaction following the one in which the snapshot was taken, the transaction is aborted by BE_1 (will be redone by FE_2). If the final query of the snapshot transaction has been executed by BE_1, then the definite outcome of the transaction is known and is returned to FE_2. If BE_1 is still waiting for the snapshot transaction to issue its final query, the transaction will be aborted after a timeout to allow FE_2 to proceed.

Migration is triggered by the client and can happen anywhere during execution of a FE. FE_1 can detect migration through an error condition on the connection endpoint, or by the forced abort of the current transaction by BE_1, at which time it can simply terminate. After migration, FE_2 imports the last snapshot from the protocol, gets the *tid* of the snapshot transaction and executes a `reconnectDB(tid)` call to the database, which returns the outcome (committed/aborted) of the transaction *tid* for which the snapshot was taken.

6.4.2 Changes to the Back-End

The back-end has been modified to spawn a new back-end process BE_2 upon reconnection of FE_2 after migration. BE_2 checks the status of the transaction which was executing at FE_1 when the last snapshot was taken. The transaction identifier *tid* of this transaction is sent by FE_2 as a parameter in the call to `reconnectDB`. If BE_2 can determine the outcome of *tid* right away, it returns it to FE_2. Otherwise BE_2 synchronizes with BE_1 and tries to obtain the result from BE_1, as BE_1 could still be executing *tid*. In response, BE_1 aborts the transaction *tid* if it is not already doing a commit on it and sends the outcome to BE_2. A global transaction identifier space is maintained across the database, along with a log of the status of aborted/committed transactions.

6.5 Writing Migratory PostgreSQL Applications

Figure 9 shows the pseudocode for a sample front-end application that handles execution of a sequence of transactions as a result of a request by a client. The front-end reports the outcome (commit/abort) of each transaction back to the client by calling `send()` on the connection. (Note that a `send` may fail due to migration; handling the error was omitted to simplify the code.) In order to report results consistently across migrations, the front-end keeps track in its exported snapshots of the point it has reached in the sequence of transactions, by recording the number of the transaction due to issue its final query in the `state_snap.last` field and its identifier in `state_snap.tid`.

After importing the snapshot of a migrated connection, a front-end resumes execution by reconnecting to the database and retrieving the outcome (`laststatus`) of last known transaction. It then reports (sends) this outcome to the client and continues execution with the next transaction in sequence after the one recorded in the snapshot.

Note that the apparently duplicate sends of a transaction's outcome from both an old and a new front-end are actually required by the deterministic execution model. When restarting, FE has no way of knowing whether the client has received the result, so it must send it. M-TCP takes care of discarding potential

```
struct {
   int tid;  // tid of last transaction to issue its final query
   int last; // logical sequence number of that transaction
} state_snap

while (accept(conn)) {
  if (import_state(conn, &state_snap, sizeof(state_snap))) {
    reconnectDB(tid, &laststatus)  // migrated connection
    send(conn, laststatus)

    goto state_snap.last + 1
  }
  else {
    receive(conn, request)         // new connection
    connectDB()
  }

  1: // start of txn 1

  txnBEGIN()
  state_snap.last = 1
  state_snap.tid  = gettxnID()
  ... do work for txn 1
  if (export_state(conn, &state_snap, sizeof(state_snap)) == EMIGRATED)
    exit()                       // take snapshot; exit if client migrated
  status = txnCOMMIT()

  send(conn, status)             // report status of txn 1 to client

  2: // start of txn 2

  txnBEGIN()
  state_snap.last = 2
  state_snap.tid  = gettxnID()
  ... do work for txn 2
  if (export_state(conn, &state_snap, sizeof(state_snap)) == EMIGRATED)
    exit()
  status = txnABORT()

  send(conn, status)

...
}
```

Figure 9: Sample front-end application using migration support

duplicates, as part of its guaranteed exactly-once delivery across migration.

6.6 An Application with Migratory PostgreSQL

We have integrated a front-end application with the Apache [1] web server to be able to test our system over HTTP. In this application the client uses a HTTP request to the web server to send a batch of transactions for execution on the database. The front-end executes the transactions sequentially and reports to the client the outcome of every transaction. We used a migration trigger policy module that uses as metric the response time and triggers migration when the response time, measured at the client side, exceeds a threshold. We simulated load on the server by inserting artificial delays before sending responses back to the client. This forces migration of the client connection to another front-end, which resumes execution and supplies the result that the client was expecting from the previous front-end. Consistency of the database is preserved across multiple migrations, and the client always gets the expected responses.

An important limitation of our prototype is that in a real web server such applications would be usually executed as CGI scripts by the server, generating dynamic content. Handling dynamic content across migration is one of the subjects of our current work in system support for migration.

Although we do not have yet a real application implemented on our Migratory PostgreSQL system, the prototype proves that the system is usable in web-based applications. Efforts are on to build a migration-aware full-fledged transactional server application. The goal of this system would be to support execution of transactions on a database from a remote client, while tolerating loss of network connectivity.

7 Limitations

In this section, we discuss several limitations of the current implementation of M-TCP. All these issues are the subject of our current research.

- **Lack of fault tolerance support.** Our current prototype assumes that the origin server of a migrating connection is still alive at the time of migration, such that the state of the connection can be extracted by the destination server. As a result, a crash of the origin server would make migration impossible. This problem is exacerbated by the fact that in the current implementation we only support lazy (on-demand) state transfer.

  Several possible solutions to this problem, leading to a fault-tolerant implementation of M-TCP, are: *(i)* use a logger to log the state associated with a connection. A problem with this approach is that logging of received data would take place on the critical path; *(ii)* store the state associated with a connection at the client, instead of keeping it at the server. Logs of data can be kept in volatile memory, and logging is done out of the critical path. The approach will incur the cost of transferring state snapshots to be stored at the client; *(iii)* perform eager transfer of state to the destination server, in combination with logging at the client. In this case the origin server would proactively transfer connection state to a destination server, i.e., in anticipation of a migration and regardless of whether it may occur or not.

- **Inter-server communication over TCP/IP**. The transfer of state associated with a connection is carried over TCP. As a result, the migration of a connection may be slowed down due to interference of other network traffic with the M-TCP state transfer at the origin server (e.g., in case the server is overloaded or under DoS attack). Because the networking software is implemented as a common protocol stack in the host OS kernel, the M-TCP state transfer traffic competes for resources in the server host (memory buffers, CPU interrupts and processing) with other network activities, including handling regular service traffic to/from clients.

  A possible solution to this problem is to use non-intrusive, memory-to-memory communication to carry the M-TCP control traffic, instead of using the IP protocol stack. Emerging standards for communication architectures like Infiniband [2] make possible remote memory operations between

hosts, using specialized IPv6 bridges. Using remote memory accesses for transfer of M-TCP state decouples service from control traffic *inside* the host[3] and also helps improving M-TCP performance by eliminating host CPU interrupts and additional buffering.

- **Single connection migration**. Migration is limited to a single connection. This means that the granularity of the service session is assumed to be one connection which is a limitation for client-server applications that may require establishment of multiple connections in parallel for a service session. In addition, the server may open other connections for servicing the client. In this case, while migrating the server endpoint, all the connections opened on behalf of the client must be also migrated.

- **Single-process server state**. The per-connection state is assumed to be confined to a single process context. This means that it is not possible to migrate a connection in case the application state spans multiple process contexts, e.g., if encapsulated in processes spawned on behalf of the client. Several such processes may be executing work at the server for a given client. A notable example is the execution of CGI scripts by an HTTP server on behalf of a client. In this case the application-specific state is distributed across the chain of worker processes. Some form of system support is needed to assemble, transfer, and reinstate it at the destination server.

8 Conclusions

In this paper we have described the use of M-TCP (Migratory TCP) [28, 26], a reliable byte-stream, connection-oriented transport layer protocol that supports efficient live connection migration, in building highly available Internet services. M-TCP enables a cooperative service model in which similar servers, possibly distributed across Internet, cooperate in sustaining service to their clients by transparently moving connections between them. Services that can benefit from such a model, and therefore can use M-TCP for high availability, typically involve long-lived connections or/and are critical to the user, such that the potential cost of migration to the client can be amortized over the lifespan of a service session.

We have implemented M-TCP in a FreeBSD kernel and presented results of an experimental evaluation showing it to be suitable for efficient support of highly available services. We exemplified with support for streaming services over reliable transport connections.

We have described how M-TCP can be used to build highly available front-end interfaces for transactional database applications over the Internet. We have designed and implemented migration support into PostgreSQL, and combined it with M-TCP to build a prototype web-interfaced transactional application.

References

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

[2] Infiniband Trade Association. http://www.infinibandta.org.

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

[4] 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.

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

[6] A. Bakre and B. R. Badrinath. Handoff and System Support for Indirect TCP/IP. In *Proc. Second Usenix Symposium on Mobile and Location-dependent Computing*, Apr. 1995.

[7] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP Performance over Wireless Networks. In *Proc. 1st ACM Int'l Conf. on Mobile Computing and Networking (Mobicom)*, Nov. 1995.

[8] 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.

[9] 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.

[3] M-TCP service and control traffic are decoupled outside the host by using distinct interfaces and networks.

[10] D. Clark and W. Fang. Explicit Allocation of Best Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking*, 6(4):362–373, Aug. 1998.
[11] E. N. Elnozahy, L. Alvisi, D. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.
[12] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.
[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.
[14] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Sheme for TCP. In *Proc. ACM SIGCOMM*, Aug. 1996.
[15] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM*, Aug. 1988.
[16] C. Krasic, K. Li, and J. Walpole. The Case for Streaming Multimedia with TCP. In *Proc. 8th Int'l Workshop on Interactive Distributed Multimedia Systems (IDMS)*, Sept. 2001.
[17] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental Study of Internet Stability and Backbone Failures. In *Proc. 29th Int'l Symp. on Fault-Tolerant Computing (FTCS)*, June 1999.
[18] C. Labovitz, G. R. Malan, and F. Jahanian. Internet Routing Instability. In *Proc. SIGCOMM'97*, Sept. 1997.
[19] D. Lin and H.Kung. TCP Fast Recovery Strategies: Analysis and Improvements. In *Proc. IEEE INFOCOMM '98*, Mar. 1998.
[20] D. A. Maltz and P. Bhagwat. MSOCKS: An Architecture for Transport Layer Mobility. In *Proc. IEEE INFOCOM*, Mar. 1998.
[21] M. Mathis and J. Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. In *Proc. ACM SIGCOMM*, Aug. 1996.
[22] J. Postel. RFC 793: Transmission Control Protocol, Sept. 1981.
[23] Y. Rekhter and T. Li. RFC 1771: A Border Gateway Protocol 4 (BGP-4), Mar. 1995.
[24] 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.
[25] A. C. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *Proc. 6th ACM MOBICOM*, Aug. 2000.
[26] K. Srinivasan. M-TCP: Transport Layer Support for Highly Available Network Services. Technical Report DCS-TR-459, Rutgers University, Oct. 2001.
[27] 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.
[28] 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.
[29] Z. Wang and J. Crowcroft. A New Congestion Control Scheme: Slow Start and Search (Tri-S). *ACM Computer Communications Review*, Jan. 1991.
[30] Z. Wang and J. Crowcroft. Eliminating periodic Packet Losses in the 4.3-Tahoe BSD TCP Congestion Control Algorithm. *ACM Computer Communications Review*, Apr. 1992.
[31] C. Yang and M. Luo. Realizing Fault Resilience in Web-Server Cluster. In *Proc. SuperComputing 2000*, Nov. 2000.