MTCP : TRANSPORT LAYER SUPPORT FOR HIGHLY AVAILABLE NETWORK SERVICES

BY KIRAN SRINIVASAN

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Liviu Iftode

and approved by

_____

_____

_____

New Brunswick, New Jersey

October, 2001

ABSTRACT OF THE THESIS

MTCP : Transport Layer Support for Highly Available Network Services

by Kiran Srinivasan

Thesis Director: Liviu Iftode

We propose a transport layer protocol designed for building highly available Internet network services. The transport layer protocol would provide a mechanism to transparently migrate the server endpoint of a live connection. The connection migration is dynamic and can happen multiple times during a client-service session. Our migration mechanism is client-initiated, integrated in a migration architecture that uniformly addresses various types of events that affect the quality of service perceived by the client. In the architecture, the migration mechanism is independent and decoupled from any migration policies. The migration mechanism can be applied for different contexts like server load balancing, to provide fault tolerance across server crashes, to improve the performance as perceived by the client etc. We examine and address a number of challenges posed by the transfer of connection state required for a connection migration to both the OS and the application layer. We describe the details of our design and an implementation, as well as experimental data that suggests the usefulness of this approach.

Acknowledgements

First and foremost, I would like to thank my advisor Professor Liviu Iftode, without whose guidance and encouragement it would have been impossible to bring my research work to its logical conclusion. His motivation and drive is primarily responsible for arousing my interest in systems research. I would also thank him for giving me this unique oppurtunity to learn and enjoy working on this project.

My heartfelt and sincere thanks go to my friend and project mate Florin Sultan. I can clearly say that without Florin's help and suggestions the project wouldnt have been what it is today. I owe him a lot for his contributions to the design and implementation of this project. I would like to thank him specially for the long hours he spent with me working on the project in the past two semesters.

I would like to thank the other members of DisCo Lab especially Murali, Aniruddha, Suresh and Deepa for their support and help in the project. I would also like to thank them for the innumerable occasions we had fun and laughter in the lab.

Last but not the least, I would like to thank my manager Mr. Thomas Truong at IBM Almaden Research Center for being considerate in providing me the time and resources to complete my dissertation.

Dedication

To my parents.

Table of Contents

List of Figures

Chapter 1

Introduction

The Internet has grown manifold in the last few years. There are millions and millions of clients and servers using this infrastructure for a variety of applications. The impact of this phenomenal growth has had some unexpected and worrisome developments. The first important development of this growth is that it has led to many new and complex network services in addition to the traditional network services. These services/applications have more advanced and demanding requirements compared to the traditional services.

We further this discussion in the following sections with a description of Internet services and the foundations they are built on. This is followed by an explanation into the need for highly available network services in the current Internet. We conclude this chapter with our approach in alleviating some of the problems faced by Internet services.

1.1 Internet Services

In this section, we describe the nature and characteristics of Internet services. We describe the changes and developments that have taken place in Internet services over the years and how each of them motivate our research work.

1.1.1 Changes in the Internet Services Model

The growth in the Internet has lead to some changes in the model and architecture of Internet services from the traditional. There has been a paradigm shift in viewing Internet resources as services rather than as servers. This means that the client (user of the resource) is no longer interested in the identity of the physical machine providing the resource. The client is primarily interested only in the nature of the service, no matter from which physical machine. In other words, this means a loss of coupling between the data stream being received from a service and

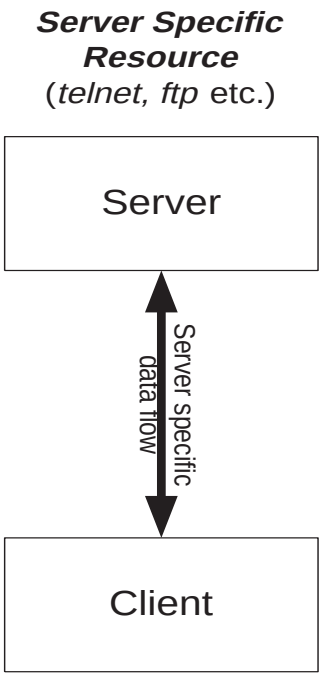

Figure 1.1: Traditional client-server model

the IP address of the exact machine providing the stream. We shall see that this is a change that can be exploited to alleviate the problems posed by the growth of the Internet.

In the traditional service model (Figure 1.1), the client is concerned about the remote entity (like `telnet service, FTP service` etc.), whereas in the emerging model (Figure 1.2) the client can receive the service from any one out of potentially many servers. In the new scenario, the client would prefer to switch between servers during the lifetime of the connection as soon as it perceives low quality of service from the current server.

The more popular and sometimes important services on the Internet are subject to overload because of millions of clients accessing these services simultaneously. There have also been numerous cases of DoS (Denial of Service) attacks on popular Internet services, affecting their availability. The net effect of both these developments is a poor quality of service perceived by clients.

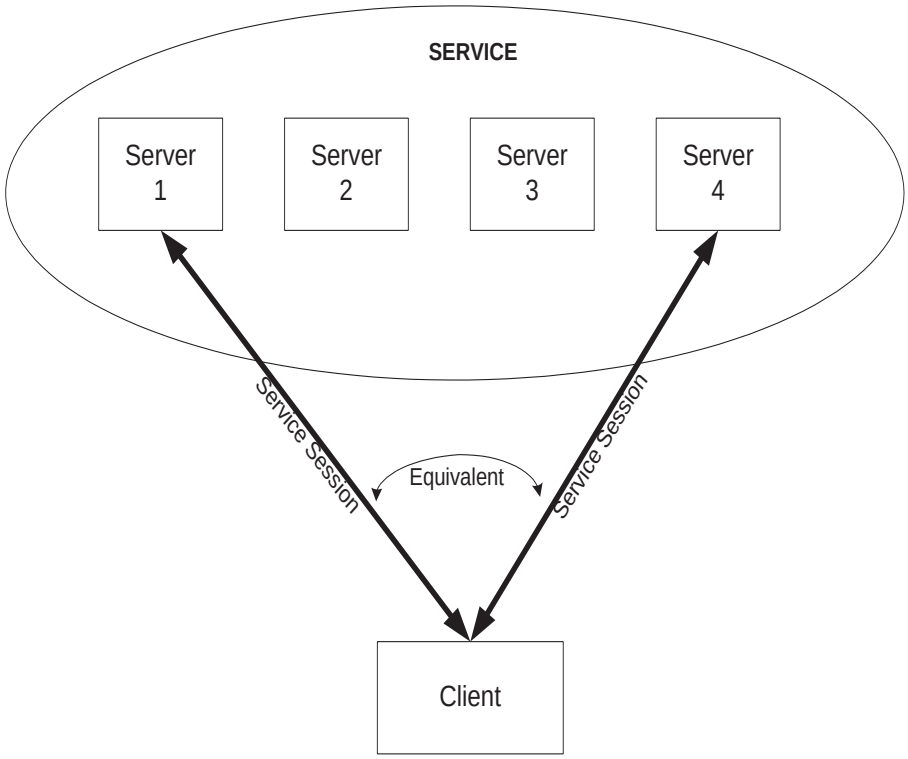

Figure 1.2: Emerging network service model

High Availability in Internet Services

An important metric that is useful in defining the quality of a present-day Internet service is *availability*. Availability helps in analysing the ability of a system to perform under conditions of failure.

There are several definitions for this term depending on the context. The most generic definition and also quite relevant in our context is:

- Availability: The probability that a system is functional at a given point in time in a specified environment.

The mathematical definition of availability is more precise:

Availability = MTBF / (MTBF + MTTR)

- MTBF = Mean Time Between Failures.
- MTTR = Mean Time To Repair (Recover).

According to this definition, if MTBF is very large (it rarely fails) compared to MTTR then the system is highly available. Even for a system that fails frequently (a low MTBF), if the MTTR is sufficiently low then the system is *highly available*. In the Internet, the second scenario is more relevant. Internet services do fail frequently and the need is to lower MTTR to provide availability.

Another useful and related metric is *reliability* which is defined as the probability that a system functions without failure for a specified time in a specified environment. From the definitions, it is clear that a highly available system need not be reliable and vice versa. A system that fails frequently but is able to recover very quickly has low reliability but is highly available. Similarly, a system that is high on reliability (it works for a very long period of time and fails very rarely) might have the characteristic that once it fails, it fails completely. Such a system is not highly available as it never recovers after a failure.

Today, a large number of important and critical services are provided by means of the Internet. The commercial interests involved in keeping a service performing continuously are of great proportions. These services need to be protected against severe conditions like sudden

spikes in the number of requests, traffic congestion, DoS attacks etc. Internet services need to constantly provide good service to the clients in order to retain them. Moreover, unavailability of some critical services like banking services could have impact on a global scale in a short period of time.

1.1.2 TCP based Internet Services

A majority of the services existing in the Internet today are built over the transport layer protocol TCP (Transmission Control Protocol - RFC 793), for example, HTTP, FTP, `telnet` etc. TCP is a connection-oriented reliable transport layer protocol. TCP is built over the unreliable, best-effort connection-less network protocol IP (Internet Protocol). TCP has features to provide flow-control between the sender and the receiver, thus enabling senders and receivers with different speeds to communicate. This is extremely important in the case of communication via the Internet as hosts connected to the Internet are heterogeneous.

The way TCP deals with network congestion is particularly relevant in the context of our work. The mechanisms used by TCP to alleviate network congestion can be classified into the following :

- *Slow start:* to enable a host to quickly learn the state of the network and the proper rate to send packets;
- *Timeouts and fast-retransmits:* to detect a packet loss in the network;
- *Congestion Avoidance:* to continuously probe the network to see if the path capacity has changed (this is done after a slow start or while recovering from a packet loss).

These mechanisms are some of the salient and noteworthy features of TCP that has led to its wide application. Since these algorithms and mechanisms were introduced there have been several efforts to improve their behavior [13, 14, 15, 16, 17, 18, 19]. The important thing to note is that in all the cases above a packet loss is being treated *as a loss of the packet in the network due to congestion*, whereas the timeout for a packet could have occurred because the packet was never accepted due to server overload or because of a DoS attack.

The basic feature of a TCP connection is that it is associated with a pair of IP addresses,

one for each endpoint of the connection. This feature of TCP leads to several shortcomings given the changes in the Internet services model:

- TCP creates an implicit *binding between the service and the IP address*. This model is overly constraining in the present day Internet, primarily because the clients are concerned mainly about the content being served rather than the exact server from which it is being served.

- The only way TCP reacts to lost or delayed segments is by retransmission. The retransmission too is to the same remote endpoint (bound by a specific IP address) of the connection. Even when the content can be provided from a different server, TCP *retransmits to the same remote endpoint*. The remote endpoint to which the retransmission is done might be down or unresponsive due to overload. The SCTP (Stream Control Transmission Protocol) [27] is an useful effort in solving this problem.

- TCP is *oblivious to the quality of service* being perceived by the upper layer protocols or applications (i.e. the client).

All these observations point to the fact that TCP is not suited for a growing class of applications in the Internet.

1.2 Our Approach and Contributions

In this section we give a brief overview of our approach in solving the problems faced by Internet services in the changed services model.

1.2.1 Approach

We have seen in the previous section the various shortcomings of TCP in the context of the new class of Internet services. *TCP does not improve the availability of a particular service in the emerging model because of the implicit binding between the service and a particular IP address*. We propose a transport layer protocol - MTCP (Migratory-TCP) that aims to lower the MTTR of a service by means of dynamic connection migration. When the service degrades or fails abruptly, our solution aims to resume the service by migrating the connection to a new

server. The migration of the connection can happen many times and at any point of time during the service session. In this context, the MTTR of a service is the time taken to migrate the connection and resume service.

This migration of the connection is totally transparent to the application attached to the connection at the client endpoint (Figure 1.2). However, the application at the server side has to be rewritten to enable it to recognize and support connection migrations. In order to make use of the protocol, the server application needs to utilize a minimal API and should adhere to a particular programming model. Overall, the effort involved at the server side to take advantage of this protocol is small compared to the benefits of high availability.

The protocol that we propose is a modified TCP, which retains all the salient and necessary features of TCP related to flow control. Moreover, our protocol has been designed to be compatible with TCP. This implies clients with ordinary TCP/IP stacks and without the ability to support migrating connections can interact with services that are built over our protocol.

1.2.2 Contributions

The main contributions of our research work are:

- We designed a transport layer protocol that enables dynamic connection migration of the service endpoint during the course of a service session.
- We have shown how our protocol can be utilized in building highly available Internet services.
- We have implemented a prototype and have demonstrated the feasibility of our approach through experimental evaluation.
- The design of MTCP was presented as a position summary [3] in the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII) held in May 2001. A more detailed version of the paper is available as a technical report [4].

The rest of the document describes the technical details of our approach. In Chapter 2, we compare and contrast against similar research work and see the benefits of MTCP versus the other alternatives. In Chapter 3, we give a detailed discussion on the design of MTCP. In

Chapter 4, a description on the implementation details of the project is presented. This chapter begins with an overview of data structures in the current protocol stack, followed by a description of our protocol and a typical connection migration scenario. In Chapter 5, preliminary performance evaluation of our mechanism follows the implementation. Chapter 6 presents the issues related to applications built over MTCP. In Chapter 7, we conclude with a discussion on the possible future directions of research related to MTCP.

Chapter 2

Related Work

Classifying broadly we have three classes of related works:

- TCP
- Fault-tolerant systems
- Process migration

A description of each of these classes and their relevance to our work is given in the following sections.

2.1 TCP Related Works

Incorporating high availability in Internet services through transport layer support has been approached in many ways. These methods include incorporating fault-tolerance in TCP [2], new protocols (SCTP - RFC 2960), and also some application-specific solutions [9]. TCP/IP connection-handoff protocols have also been used in the context of mobility extensions to TCP/IP [6, 7, 8] . We describe each of these approaches in the following sections.

2.1.1 Fine-Grained Failover using Connection Migration

This is a fail-over architecture that enables migrating HTTP connection endpoints within a pool of support servers that replicate per-connection soft transport and application state [9]. In their case, the connection migration is initiated only by the new server, on events like the origin server's failure or overload. Their architecture adds a HTTP-aware module at the transport layer that extracts information from the application data stream to be used for connection resumption. While the design leads to no change in the server application, it is not general, as the transport

layer must have knowledge of the upper layer protocols. The design depends on the information in the application data stream being:

- sufficient for describing application state associated with a connection,
- compatible or up-to-date with respect to the in-kernel module, and
- always accessible, i.e. application traffic can only be (unencrypted) clear text.

In our scheme, we expect the server application to change, to assist a transport layer protocol that allows for dynamic connection migration. We allow the server to specify to the kernel the state needed to resume the data transfer in case of connection migration. This application-specific state is completely opaque to the kernel and the transport layer protocol.

2.1.2 Stream Control Transmission Protocol (SCTP)

The SCTP (RFC 2960) is a transport layer protocol aimed at solving some of the problems faced by Internet services due to TCP. The key features of this protocol in comparison to TCP are:

- Sequenced delivery of user messages within multiple streams.
- Support for multi-homing (many IP-addresses per endpoint) at either or both ends of an association.

This approach aims at providing high availability to a service by allowing multiple connection streams for a single service session. Network-level fault tolerance is offered by means of multi-homing. The motivation for SCTP is identical to ours and is to alleviate the shortcomings of TCP for the emerging services. The difference is in the approaches towards achieving high availability. Multi-homing is significantly useful only when the network interfaces are connected to totally different networks to shield against network congestion. This is analogous but different from our approach of providing high availability through connection migrations which doesn't face such requirements. Our migration mechanism can also be built over SCTP and would enhance the features of SCTP.

2.1.3 Fault-Tolerant TCP (FT-TCP)

FT-TCP [2] is a scheme for transparent (masked) recovery of a crashed process with open TCP connections. A wrapper around the TCP layer intercepts and logs reads by the process for replay during recovery, and shields the remote endpoint from failure. Their system focuses on masking a fault, while we address fine-grained light-weight migration of a connection endpoint. Although their scheme could be used for connection migration, this would require coarse-grained migration of the full process context. This research work does not address the issue of availability in the changed services model. Our system allows migration of only a connection endpoint, whereas the process can continue to run. The emphasis in FT-TCP is on fault-tolerance and not on providing high availability to services. This is so because the time to recover a particular connection can be high as reincarnated connections need to start from the beginning.

2.1.4 TCP Connection Handoff in Cluster-based Web Servers

The Location Aware Request Distribution (LARD) project [5] proposes TCP connection handoff in clustered HTTP servers for distributing incoming requests from a front-end machine to server back-end nodes. It is limited to a single hand-off scheme, where a connection endpoint can only migrate during the connection setup phase, whereas in our solution connection migration can happen dynamically and at any time during the course of the connection. Multiple hand-offs of persistent HTTP/1.1 connections are mentioned only as an alternative, but no design or implementation is 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 comparison to our scheme, the TCP connection handoff does not work between servers distributed across a WAN, and hence cannot be used to alleviate network congestion between the client and the cluster-based server.

2.1.5 Layer-4 Switches

There are many Layer-4 switches [34, 35] available commercially in hardware. The idea is for a front-end switch to distribute connection requests among a pool of back-end servers. The

switch does this by inspecting for the TCP SYN segments within an IP datagram. The handoff to the back-end server is done at the time of connection establishment only. This approach doesn't shield the client from a server performance degradation during the course of the session after establishment. In contrast, our mechanism provides for dynamic connection migration at any point in the session. Thus, in the event of server performance degradation, we can migrate the connection to a better performing server.

2.1.6 MSOCKS

The emphasis of this project is on providing continuity in service sessions to mobile clients when they are equipped with multiple interfaces [23]. It is a proxy based solution where the client is connected to the proxy with more than one network link. 'Splicing' of TCP connections is done at the proxy (by means of sequence numbers mapping) to enable a 'session' to move from one network of the mobile client to another. Though the project addresses the management of a service session by more than one TCP connection, it is not concerned with the problem of enabling high-availability in services. In relation to our work, this project does not deal with migration of the server connection endpoint. Hence, the problems of transfer of connection state and state synchronization at the server endpoint do not appear in this context, while these are the problems we focus in our work. In summary, our solution aims to enable building of highly available services while the focus of MSOCKS is to help mobile clients maintain connectivity to services through a proxy. The two solutions are complementary and we could think of MSOCKS operating over MTCP.

2.1.7 Mobile TCP/IP Related Works

The mobile TCP approaches do not consider the task of migrating the connection endpoints between physically distinct machines. Thus they do not cater to the need of services in the emerging Internet services model where the client is not concerned about a particular server providing the service. 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 after proper authentication [8]. These approaches might enhance the availability of a service in the face of network congestion but

cannot do so when the server is overloaded.

2.2 Fault-Tolerant Systems

Log-based rollback recovery is a very familiar and commonly used technique employed in many fault-tolerant systems [36]. The NonStop kernel [37] was the first system to employ this technique. In our protocol, we use this technique to synchronize the per-connection application and protocol states. The state snapshots used in our protocol are analogous to *checkpoints*. Logs are maintained in the protocol for replay after rolling back to the last state snapshot while resuming service after migration.

2.3 Process Migration

In our protocol, we resume a service session at the destination server using the state information of the server application process at the origin server. This involves techniques related to some aspects of process migration. All process migration related projects deal with a heavy-weight application process migration, where the entire server application process is migrated [43]. Condor [40] is a software library package that enables user-level process migration. Examples of operating systems that have process migration built in are: Locus [38], MOSIX [39], Sprite [41], NOMAD [42] etc.

Chapter 3

Design of MTCP

This chapter explains the key design issues involved in dynamically migrating connections across the WAN. At the outset, our basic migration model and architecture are described. This is followed by a section on the connection migration mechanism elaborating on the design problems and solutions. Finally, a section on the potential policies that can to be used with this mechanism is presented.

3.1 Migration Model

In our model, an Internet service is represented by a set of *geographically dispersed servers* that provide the same service (Figure 3.1). A client is connected to one of the many equivalent servers. The complete client interaction with the Internet service from initiation to termination represents a service session.

In our model, connection migration involves only one endpoint of the connection (the server side), while the other endpoint (the client) is fixed. At any point during the service session, the client might suffer loss in the quality of service on the connection associated with the current server. This might be due to several reasons like network congestion, server overload, DoS attack on the server etc. This will trigger the client side transport layer protocol to migrate the connection to a better performing equivalent server as shown in Figure 3.1. The old and the new servers *cooperate* to facilitate the connection migration. The connection migration is dynamic and can happen multiple times during the session.

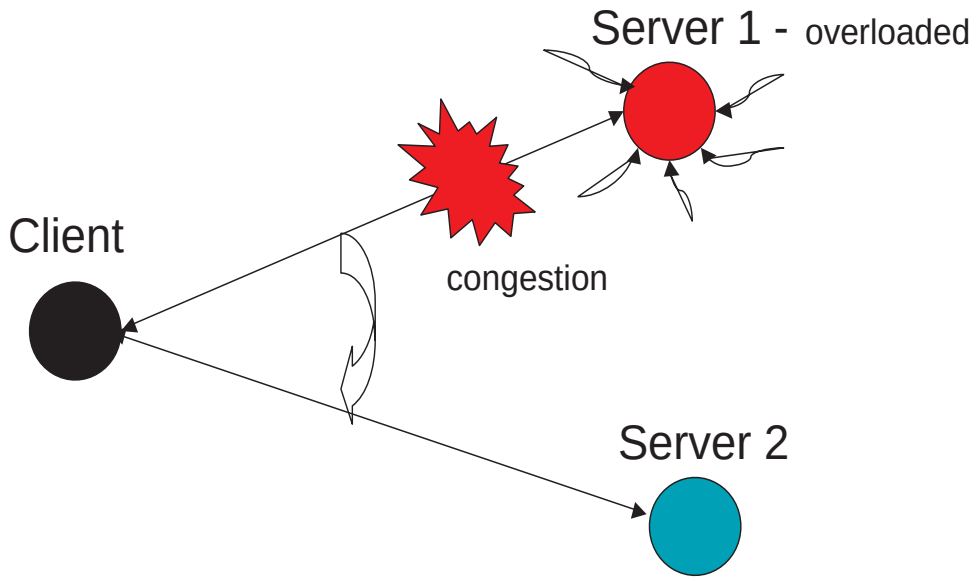

Figure 3.1: Migration model

3.2 Migration Architecture

In this section we present the architecture underlying our connection migration mechanism. We do not specify any policy or implementation related issues like when to migrate, what events trigger a migration, how/when is the connection state transferred from the origin to the destination server etc.

The mechanism for connection migration reincarnates the migrating endpoint of the TCP connection at the destination server and also establishes a *restart point* for the server application. The *restart point* represents a point in the service session which can be used as a reference by the new server to resume the service in a consistent manner. The migration mechanism performs these actions by transferring the state associated with the connection at the old server to the new server. In describing our model, it is not essential how the connection state is transferred to the destination server:

1. The state might be eagerly propagated from the originating server to the new server(s) (at regular intervals, each such instant corresponding to a *restart point*).

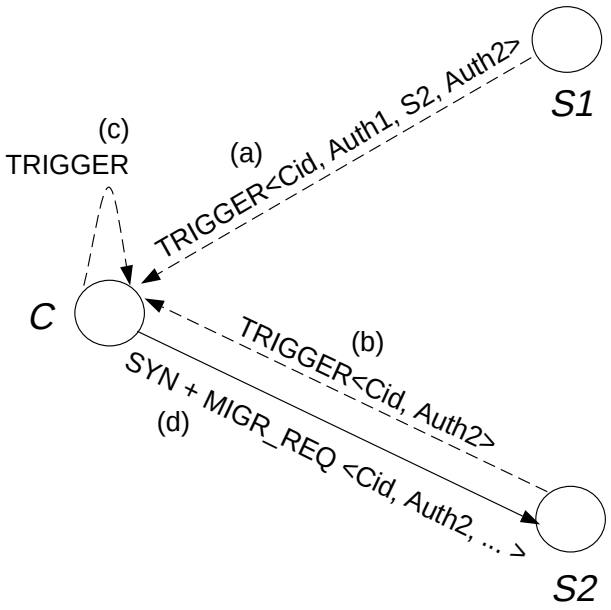

Figure 3.2: Triggers and Initiators

2. The state can also be transferred on demand, i.e. ”lazily” , (at the request of the destination server, when the migration is done oblivious to the originating server). The new server might ”pull” the state from the originating server.

3. The state might also be stored at the client and provided by it when initiating a migration.

However, these methods differ in implementation. The kind of benefits accrued by our migration model is also closely linked to the method in which the state is transferred.

Our architecture does not place restrictions on how the transfer of state takes place : through a TCP/IP socket, by means of memory-mapped communication like VIA etc. The architecture is intended to be a general one and our mechanism within this architecture can be uniformly used to react to various events that may require a connection to migrate, regardless of where the decision to migrate is taken. For example, a load balancing decision is always taken at the server side, while migration as a result of perceived latency is always taken at the client side.

3.2.1 Triggers and Initiators

Figure 3.2 shows the three entities involved in a migration: the origin server S1, the destination server S2, and the client C. Recall that one of our goals is not to change client applications, therefore the levels at which the three entities operate are not symmetric: the entity labeled C actually represents actions supporting the transparent migration of the remote endpoint that take place in the transport protocol layer on the client process's side of a connection. However, actions of S1 and S2 may originate either at the transport or application layer.

The basic idea in our architecture is to distinguish between the two entities: *trigger* and *initiator* of a migration. The trigger decides that the migration of the remote endpoint of a client connection is required. The initiator is the entity that actively starts the process of reinstating the connection at the destination server. In our architecture, the trigger of the migration can be any of the three parties involved. However, the sole initiator of the migration is always the client C. The role of trigger is just to inform the initiator (the client) that it must migrate a certain connection in which it is involved, while the role of the client is to actively take the steps to do so.

The advantage of having only the client initiate the migration lies in using a uniform mechanism, regardless of which of the parties triggers the migration. When an event that prompts a connection migration occurs, the trigger (if remote) sends a special TCP segment to C, with an option *TRIGGER = (Cid, Auth_trig, (S_dest, Auth_dest))* to signal a request for migration (case (a) or (b) in Figure 3.2). The option carries the 4-tuple connection identifier *Cid* (IP addresses and ports for both C and S1 endpoints), along with authentication information *Auth_trig* that identifies the trigger as legitimate for asking the client to initiate a migration. In addition, it may contain the destination server's IP address and an authenticator for the migrating connection at the new server (for example, if the trigger is S1, case (a) in Figure 3.2). The segment carrying the *MIGRATE_TRIGGER* option can be sent with a special SYN segment on the port used by the Cid (the option itself discriminates the SYN for special processing).

3.3 Migration Mechanism

In this section, we describe the design of the migration mechanism. The key design problem that we face is that of identifying the state that needs to be transferred across cooperating servers. In addition, resuming a service session at the new server from the point it left off at the old server raises several interesting questions. These questions and their solutions are dealt with in the following subsections.

3.3.1 The Problem

The primary question is the nature and the contents of the *state* to be transferred between the servers that cooperate to migrate a connection. As the TCP/IP stack is implemented in the OS, migrating a live TCP connection requires transfer of its associated protocol state (e.g., sequence numbers information, buffered data of unacknowledged segments etc.) from the origin (old) to the destination (new) server. The protocol state of the connection transferred to the new server should be sufficient to create a connection endpoint at the new server. In addition, as most Internet services are stateful, we can think of an application-level state associated with a connection that also needs to be transferred.

The existence of an *application-level state* raises the question of defining the contents of this state. The state reached by the server process while servicing a particular client's session cannot be inferred from the in-kernel protocol state of the TCP connection used by the session. In addition, the new server application is expected to resume service to the client from the point it left off from the old server. As a result, the *application-level state* should describe a *restart point* in the service session. This ensures that the new server application continues servicing a session from a well-defined, application-level state.

The in-kernel data buffering done by TCP and the active read/write model of communication poses another problem for the transfer of state. The buffering causes a loss of synchronization between the protocol state and the application state. However, exactly-once semantics and reliable delivery of data should be preserved when the new server resumes service on the migrated connection. Hence there is a need to synchronize the server application state associated with the session and the in-kernel protocol state of the connection used for data transport by

that session.

3.3.2 Approach to the Problem

The kernel is oblivious to the application-specific state that a particular server application maintains for a given client session. Only the server application knows the contents and characteristics of this state. We propose a minimal interface to the OS in the form of a few system calls that can be used by the server application for exporting/importing a *state snapshot* of the server application state to the kernel. We call it a *state snapshot* because this state might continuously change with time during the course of the service session.

Our approach to these problems is best described as a contract between the server application process and our transport layer protocol (MTCP). The contract is defined as follows:

- **Server application's terms**

    1. Should **define** a fine-grained per-connection application state.
    2. Should **export** per-connection state snapshots periodically to the transport layer protocol when the per-connection application state is consistent with the data read/written from/to the buffers.
    3. After the connection migration, the destination server application should be able to resume the service after **import**ing the last per-connection application state snapshot taken at the origin server.

- **Transport protocol's terms**

    1. Should **transfer** the per-connection state to the destination server.
    2. Should **synchronize** the per-connection application state and the protocol state.

To enable the server application to follow the terms of the contract MTCP provides a minimal interface. The following are the two primitives in the interface:

1. **export_state** : Enables the server application to export the application state snapshot to the protocol.

2. **import_state** : Enables the server application to import the application state snapshot from the protocol.

If the server application follows all the terms of the contract, MTCP promises to enable dynamic connection migration of the server endpoint without loss of reliability in the data stream. Transferring the per-connection state (application and protocol) to the destination server without the server application's involvement is a simple task for the transport layer protocol. The task of state synchronization is more complex and also determines the appropriate segment of data in the socket buffers to be transferred.

In our approach, we make a clear assumption that the server application can associate a fine-grained per-connection state with every connection. Any other remaining state that is needed to resume service for this session is deemed to be accessible to the new server application. We see that this assumption holds for a sufficient number of popular server applications after certain modifications.

The *state snapshot* completely describes the point a server has reached in an ongoing session with the client and thus can be used as a reference point or restart point for resuming a connection after a connection migration. For example, consider a Web server serving static HTTP requests. A client makes a request for a particular file. In this case, the state snapshot at a particular point in time would consist of two components: the *name of the file* requested and the *file offset* denoting the point in the file up to which the server has served. We can observe that this information is necessary and sufficient for any other cooperating web server to resume the service to the client.

The most recent application state snapshot represents the *restart point* of the service session for the new server and along with the in-kernel state of the connection, enables the new server to:

1. Restart servicing the client session from the state snapshot on.

2. Replay reads executed at the old server and for which data can no longer be supplied (retransmitted) by the client-side TCP, i.e. the reads that have already been acknowledged.

3. Replay writes that were executed by the old server before migration and which are not guaranteed to have reached the client (the acknowledgments for these writes have not

been obtained) but are reflected in the state snapshot of the old server (so they cannot be re-executed at the new server).

Another key assumption in our mechanism is that the server application follows a *deterministic process execution model*. We see that this assumption is not overly restrictive and does hold for many important Internet services. The above three guarantees coupled with this assumption ensure that the new server can resume service to the client consistently after migration.

Sometimes it would be necessary to replay the reads executed at the old server after the most recent state snapshot was taken. For these reads the kernel would have discarded the data from the buffers. In order to make them available for replaying after migration we have to log these reads in the kernel. These logged reads are part of the kernel state to be transferred to the new server. The size of this state is dependent on the frequency of state snapshots taken by the server application for a particular session: the greater the frequency, the smaller the amount of logged reads. However, each time a snapshot is taken it involves an overhead due to copies of the application state and other values. Another observation in this context is that the size of this state component can be regarded as small, as in a typical server-client session the bulk of the traffic is from the server to the client.

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

We can also extend our current scheme by including in the application-level state of a connection, beside local server resources, the other connections established by the server for servicing the client. In that case, while migrating the server endpoint all the connections opened on behalf of the client must be also recursively migrated.

In summary, the distinguishing features of our approach are:

1. It is application independent. Compared to schemes like [7,8] it is general and flexible, in that it does not rely on knowledge about a given server application or application-level

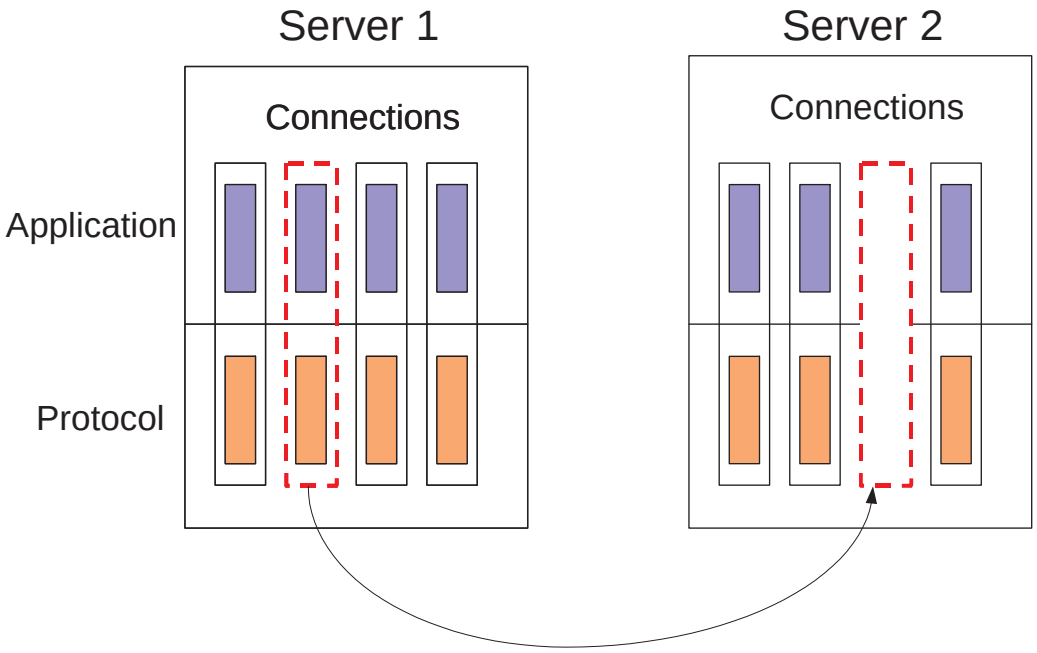

Figure 3.3: Transfer of the application and protocol states of a connection

client-server protocol.

2. It is light-weight. Compared to TCP fault-tolerance schemes like [2], which work only for a whole process context, our mechanism operates at the granularity of individual connections. It thus can be used in a TCP fault tolerance scheme where individual connections can be recovered separately after a failure, possibly on different servers.

3. It is symmetric with respect to and decoupled from any migration policy. Compared to approaches like [7], it enables both the client and server to trigger the migration, and it enables a server to control the migration process.

3.3.3 Connection State

This section presents the different components of the connection state and their dependencies. The dependencies create a need for state synchronization which is explained in the following subsections.

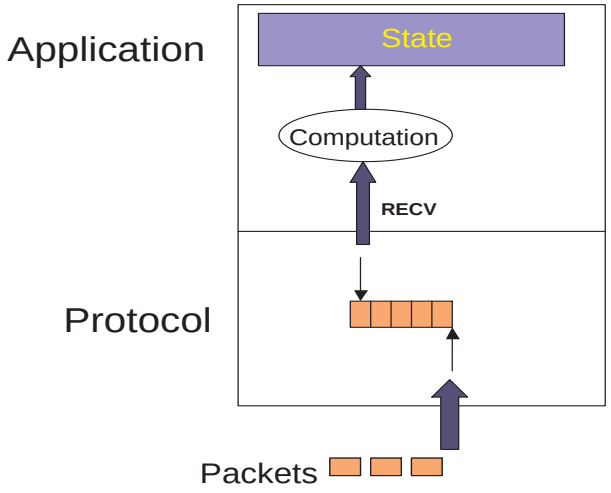

Figure 3.4: Dependency between application and protocol states

Components of the State

Classifying broadly, there are two components of the connection state, the kernel-level component and the application-level component.

Each server host has a set of connections, each one of which has a kernel-level state and an application-level state (Figure 3.3). When we migrate one particular connection (as shown in the figure) both these components need to be *reinstated at an accommodating peer*. These two components are not completely independent of each other.

There are a number of actions that take place when a packet arrives on a connection for an application (Figure 3.4). First, it changes the TCP state, i.e. the packet payload leads to changes in the sequence numbers and also gets buffered in the socket buffers. Once the application makes a read on the socket corresponding to that connection, the data moves from the kernel to the application level. Usually the data in the packet is assimilated in some way, which is illustrated as *computation* in the figure. This computation leads to changes in any application level state that an application maintains per connection. The various steps of data processing; from packet arrival until the point the data in the packet leads to changes in the application state forms a *pipeline*. As stages of the pipeline operate asynchronously, we have the problem of synchronization between the application-level and kernel-level states.

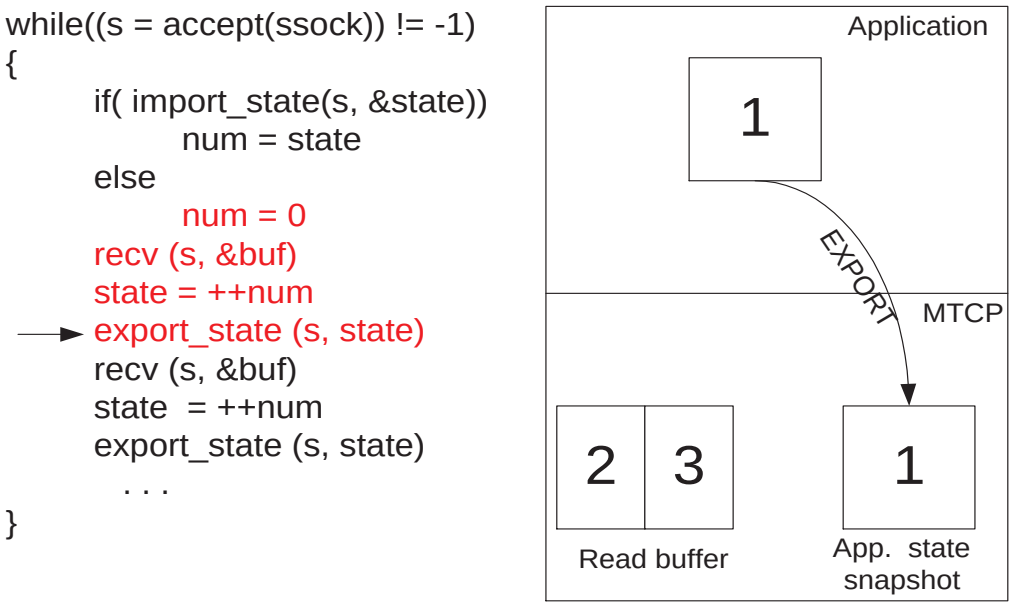

Figure 3.5: State synchronization example - I

This problem of state synchronization and its solution is very implementation specific and we describe a solution for a lazy migration scheme. In a lazy migration scheme, the new server requests the state from the old server only when the client initiates a migration to the new server. In Figure 3.4, suppose this pipeline is in operation at the origin server when a request for the connection state from the new server comes (the old server does not know when the client makes a migration request to the new server). At this point of time, the application-level state is being constantly modified, the data is still being assimilated (i.e. processing of the data is not yet complete) and new data is being buffered in the socket buffers.

Suppose that we satisfy the state request from the new server with the current application-level state and the current socket buffers in the kernel. In this case we have an application-level state that is partially modified (because the computation was going on when the request came) and which is not consistent with the data read from the kernel (as some of the data is still in the computation stage of the pipeline and not represented in the application-level state). Thus reinstating the application state and the protocol state without synchronization might lead to violation of the exactly-once semantics of TCP.

One approach to the solution would be to have access to all the data in the computation stage

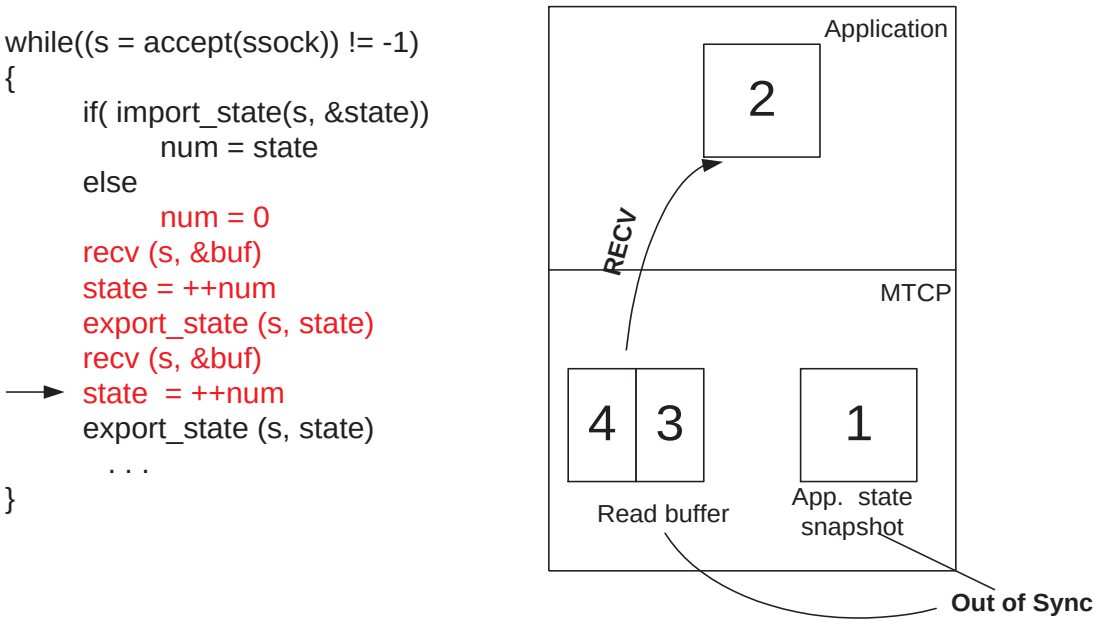

Figure 3.6: State synchronization example - II

and transfer that as part of the state. If this information is recorded in the state, the amount of state to be transferred becomes huge, as this may amount to migration of the entire process that created the connection. This is not feasible as we aim our mechanism to be applicable for migrating connections between geographically dispersed servers connected by a WAN.

Log Based State Synchronization

In the contract between the server application and the transport layer protocol described in subsection 3.3.2, one of the terms to be followed by MTCP is to synchronize the per-connection application state with the protocol state. The state synchronization problem is explained with the help of an example.

In Figure 3.5, on the left half of the figure is a simple server application program that adheres to the contract by using the primitives described. The server application reads a fixed number of messages from the read buffer. The application-level state pertaining to a particular connection at any point in time is the number of messages read. On the right of Figure 3.5 is a graphical depiction of the connection's state after the statement pointed by the arrow has been

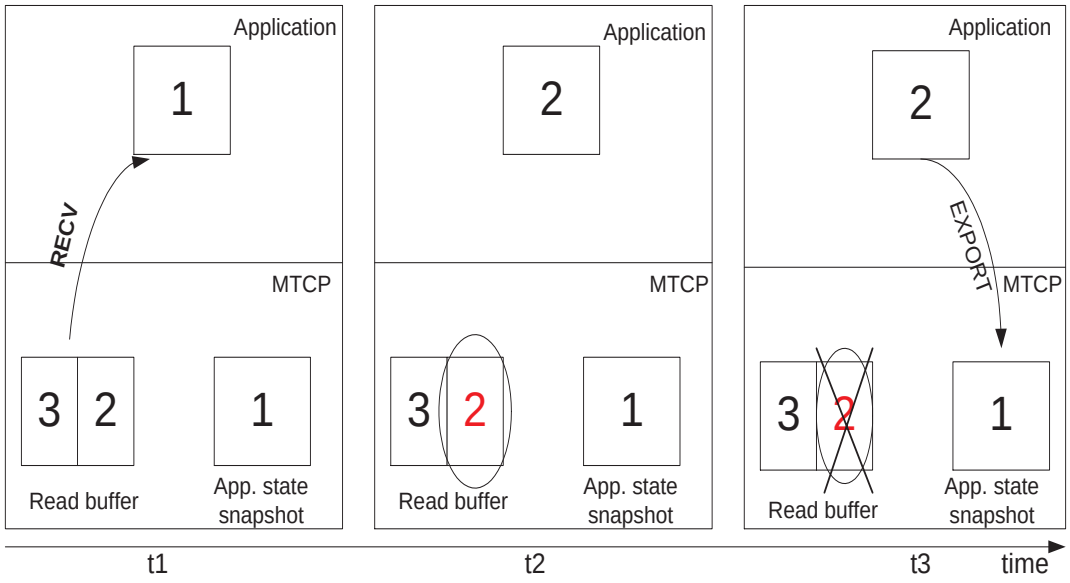

Figure 3.7: State synchronization solution

executed for a new client connection (non-migrated). The in-kernel copy of the application state snapshot, the read buffers containing the outstanding messages (2 and 3) are shown in the bottom part of the figure.

Figure 3.6 shows the state of the connection after the next two statements following the `export_state` call are executed (pointed by the arrow in the figure). Suppose there is a request for this connection's migration from the transport layer protocol of a peer at this stage in the service session. Even if the transport layer protocol transfers the per-connection application and protocol states (data in the read buffers), the server application at the destination server cannot resume the service in a consistent manner. The last application state snapshot shows that message-1 has been read, so the server application expects to read message-2, but the read buffers do not contain message-2 (only message-3 and message-4 are present). In short, there is loss of synchronization between the application and protocol states.

Figure 3.7 shows our approach in solving the state synchronization problem. The figure describes the state of the connection at three increasing points in time t1, t2 and t3. At time t1, the application has made a **recv** call, and has read message-2. Suppose the protocol *logs* this message instead of deleting it from the buffer. At time t2, the application has changed the state

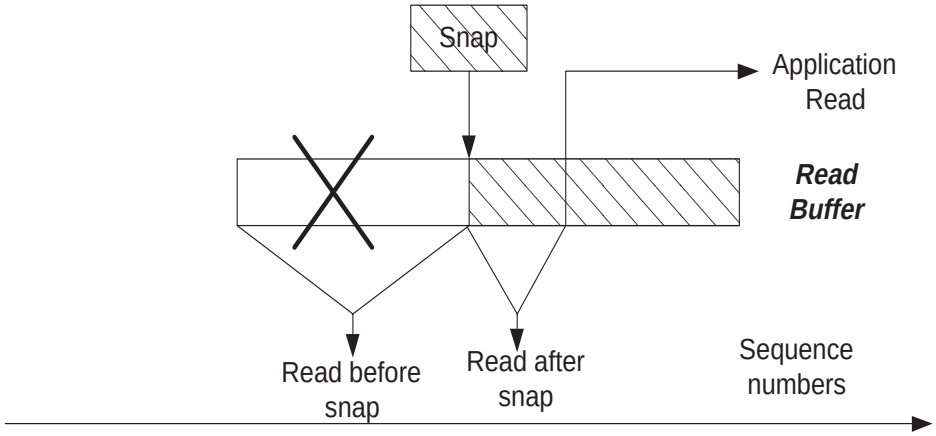

Figure 3.8: Read buffer synchronization

to reflect that it has read message-2 but it has not exported this changed state to the protocol. The scenario at t2 is similar to the one described in Figure 3.6. Suppose a migration request occurs at this point in time and the transport layer protocol transfers both the logged message and the read buffer to its peer i.e., both messages 2 and 3, as part of the read buffer state. Now, the destination server application can resume service in a consistent manner because message-2 is still in the read buffer.

Effectively, we are rolling back in time to the point the last application state snapshot was taken when we resume the service session at the destination server. Thus, our protocol guarantees state synchronization using a *log based rollback recovery technique* which is a familiar technique employed in many fault-tolerance systems.

At time t3, the application exports the changed state to the protocol. Now the application state snapshot (which reflects that message-2 has been read) is consistent with the data in the read buffers and the log can be erased. Thus we maintain the log for the read buffer data only during the interval between application state snapshots. State synchronization involves more details than the brief overview presented in the introduction above. Synchronization of the connection's read and write buffers with the corresponding application level state ensures reliable delivery of data and the exactly once semantics of TCP. Reconciling the protocol-level state at the client and new server involves reinstating all the data structures related to a connection in the kernel: socket, the protocol control blocks. Details will be provided in

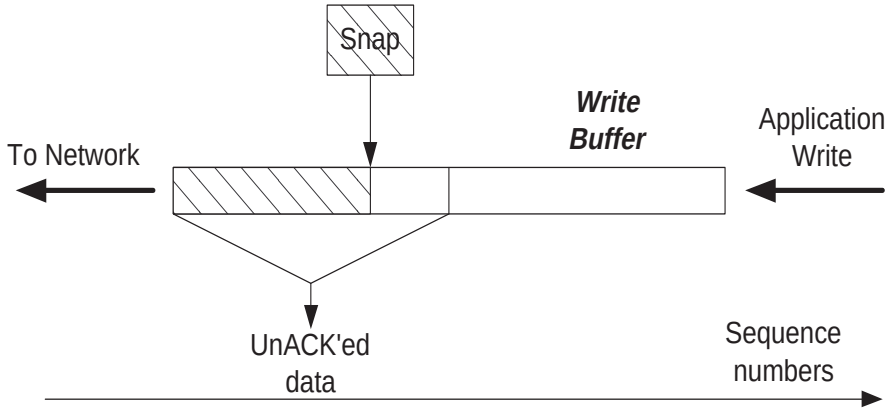

Figure 3.9: Write buffer synchronization

Chapter 4.

Figure 3.8 explains the read buffer synchronization. The *snap* in the figure represents the last state snapshot taken by the application. The shaded region shows all the data that needs to be transferred as part of the read buffer state. This region includes all the data that have been ACKed in the read buffer starting from the point the snapshot was taken. The significance of taking only the ACKed data is that the unACKed data would be present in the client's send buffer and need not be transferred. The data between the snapshot and the application read pointer in the figure represents the logged data.

Figure 3.9 explains the write buffer synchronization. The shaded portion in the figure represents the amount of data to be transferred for the synchronization of the write buffers of the client and new server. The region includes that portion of the unacknowledged data that is in the buffer up to the point the snapshot was taken (Note: Only unACKed data is present in the buffers). The rest of the buffer need not be transferred: after the state transfer, the new server application starts from the snapshot and the data after the snapshot will be regenerated as the execution unfolds.

An interesting scenario happens when the snapshot lies far to the left of the start of unacknowledged data (Figure 3.10). This scenario implies that a part of the data written to the write buffer after taking the snapshot has been acknowledged by the client. This means that some of

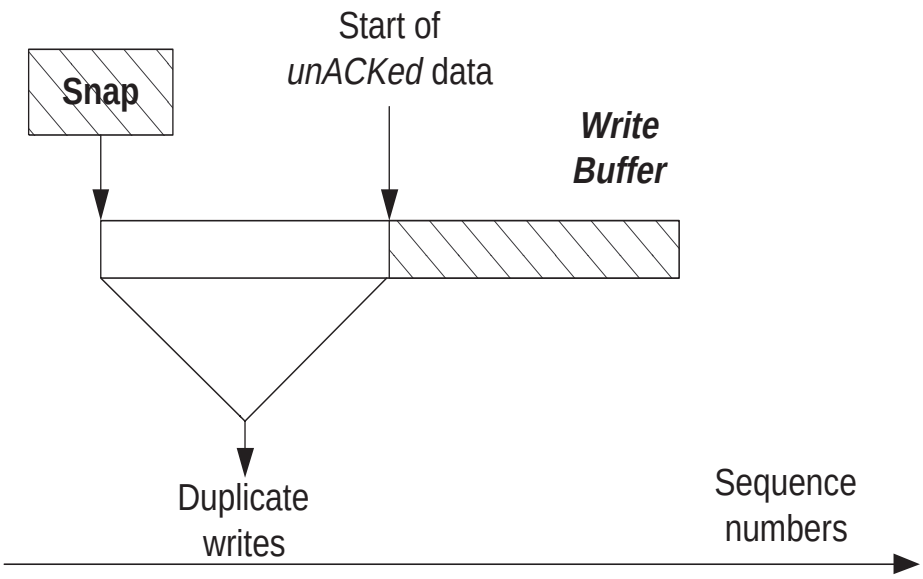

Figure 3.10: Duplicate writes to the buffer

the data (indicated the figure as *duplicate data*) will be written again during the execution starting from the snapshot at the new server. This data already has been acknowledged by the client and should not be sent again to maintain exactly-once semantics; instead it must be discarded by the kernel at the new server. In this scenario, no data in the write buffers is transferred from the old server to the new server from the write buffer. However, the number of duplicate bytes need to be computed and sent to the new server. We will explain more about this in Chapter 4.

3.4 Migration Policies

In this section, we present some of the potential policies that could be employed with our connection migration mechanism. In our design, policy decisions are decoupled from the migration architecture, including the events that trigger a migration (a timeout, a segment arrival etc.). This is an area of interesting research that needs to be explored.

These policy decisions are largely application-dependent. It is beyond the scope of our current work to define such policies. The performance evaluation in Chapter 5 explains a sample policy that we designed for experimental purposes. However, it would be interesting to suggest policies for certain classes of applications. For example,

1. *Soft real-time applications:* These applications are delay sensitive; so delay in receiving the next in-sequence segment would be an appropriate metric for designing policies in this context.

2. *Critical applications:* For critical applications, like those that involve bank transactions etc., the response time perceived at the client is the metric to be used while deciding on policies.

3. *Throughput-sensitive applications:* In this case, we need to monitor the throughput constantly, and this could be used as a metric in policies. An appreciable decrease in throughput over a period of time would warrant a migration.

Chapter 4

Implementation of MTCP

This chapter elaborates on the implementation details of MTCP. MTCP was built by modifying TCP without any changes to the core functionalities in TCP. The semantics of TCP have largely been preserved intact. The implementation of our migration scheme is highly dependent on the method of state transfer. In our current implementation, only the client acts as the trigger for a connection migration. Extending this implementation to include the servers as potential triggers is fairly simple. The method of state transfer from the old server to the new is *lazy*, whereby the new server pulls the state information pertaining to a connection from the old server on demand after the client initiates a migration. Implementations using other modes of state transfer have also been investigated.

Our current implementation is a *lazy* scheme where the necessary state is not transferred until it is required at the new server. The whole process is explained in Figure 4.1 Each message/segment in the figure is accompanied by a number in brackets which signifies the relative order in time of these messages/segments starting with the lowest (0). Once the client initiates a migration, with a special SYN to the new server, the new server requests the original server for the state. The original server then responds to that request with the state information of the migrating connection. Once the new server has the complete state information of the connection from the original server, it is assured that it can accommodate the client and hence replies the SYN of the client with an ACK accompanied by its own SYN.

The *lazy* implementation assumes that the original server is up and running to provide the state when the new server requests. Furthermore, there is an assumption that the original server is not loaded enough to reply to the new server with the state information. The transfer of state is through another TCP/IP connection between the servers termed the *control connection*. We

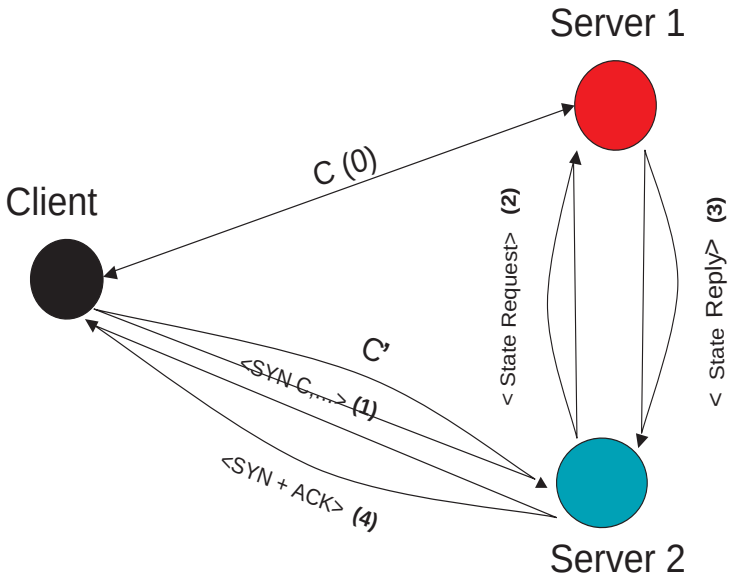

Figure 4.1: Lazy connection state transfer implementation

require a TCP connection for the control connection because we do not assume anything regarding the location of the cooperating servers. They may be geographically dispersed. Moreover, we require reliable transfer of data over this channel that TCP provides.

This implementation can be used for dynamic load balancing among a group of cooperating servers by means of connection migrations. However, this implementation wouldn't assist us in providing fault-tolerance. As fault-tolerance would necessitate the ability to salvage connections even when the original server is down.

4.1 Overview of the Kernel Data Structures used

The network protocol stack used for the implementation of MTCP is the common reference implementation of TCP/IP from the Computer Systems Research Group at the University of California at Berkeley. This has been distributed with the 4.x BSD operating system. The official name of the software is the 4.4 BSD-Lite distribution, its also referred to as Net/3. The open source operating system (that distributes the 4.4 BSD Lite source code) that was used in the project is FreeBSD (http://www.freebsd.org).

4.1.1 Primary Data Structures and their relationships

The socket and the PCBs (Protocol Control Blocks) are the primary data structures relevant in our discussion and are modified to a large extent in the implementation of MTCP.

4.1.2 The socket Structure

A socket structure represents one end of a communication link and holds or points to all the information associated with the link. This includes the protocol to use, state information for the protocol (which includes source and destination addresses), queues of arriving connections, data buffers and option flags. The following are the important and relevant fields of the socket structure:

1. so_type: This field identifies the communication semantics to be supported by the socket and the associated protocol. For TCP it would be SOCK_STREAM.

2. so_options: It's a collection of flags that modify the behavior of a socket. Some additional flags were introduced in addition to the default ones in the implementation of MTCP.

3. so_state: Represents the internal state and additional characteristics of the socket. The implementation of MTCP introduced some additional socket states.

4. so_pcb : This field points to a protocol control block that contains protocol specific state information and parameters for the socket. Each protocol defines its own control block structure, so so_pcb is defined to be a generic parameter. In the case of TCP, the control block structures are struct inpcb (for IP specific information) and struct tcpcb (for TCP specific information).

5. so_proto: This field points to the protosw structure of the protocol selected by the process during the socket system call.

6. so_error holds an error code until it can be reported to a process during the next system call that references the socket. In MTCP, this field is also used to report to the application that a particular connection has migrated.

7. `so_upcall` and `so_upcallarg`: `so_upcall` is a function pointer, and `so_upcallarg` is designed to be a parameter to that function. Whenever important events happen, such as a segment arrival, the `so_upcall` function is invoked with the `so_upcallarg` as parameter. In the implementation of MTCP, there arises a need to build in-kernel listening sockets and we use the same fields for that purpose.

4.1.3 Protocol Control Blocks

As mentioned earlier, each protocol maintains state and information in its protocol control block. Some protocols like UDP, have very little state that they do not have the need to keep a separate control block and use the same control block as IP (Internet Protocol Control Block). The protocol control blocks of relevance to our implementation are the Internet Protocol Control Block (`struct inpcb`) and the TCP Control Block (`struct tcpcb`).

Internet Protocol Control Block

The Internet PCB contains the information common to all UDP and TCP endpoints: foreign and local IP addresses, foreign and local port numbers, IP header prototype, IP options to use for this endpoint, and a pointer to the routing table entry for the destination of this endpoint. The TCP control block contains all the state information that TCP maintains for each connection: sequence numbers in both directions, window sizes, retransmission timers and the like.

Figure 4.2 summarizes the protocol control blocks that we describe and their relationship to the `file` and `socket` structures. The `socket` and the `inpcb` structure point to each other by means of the `so_pcb` and the `inp_socket` members respectively. The `inpcb` structure also contains the (local address, local port, foreign address, foreign port) tuple for this endpoint. The TCP control block `struct tcpcb` stores TCP specific information for this connection. The `inpcb` and the `tcpcb` structures also point to each other.

TCP Control Block

The TCP control block is a very large and complex structure. Both the IP control block and the TCP control block have been modified for the implementation of MTCP. The modifications and

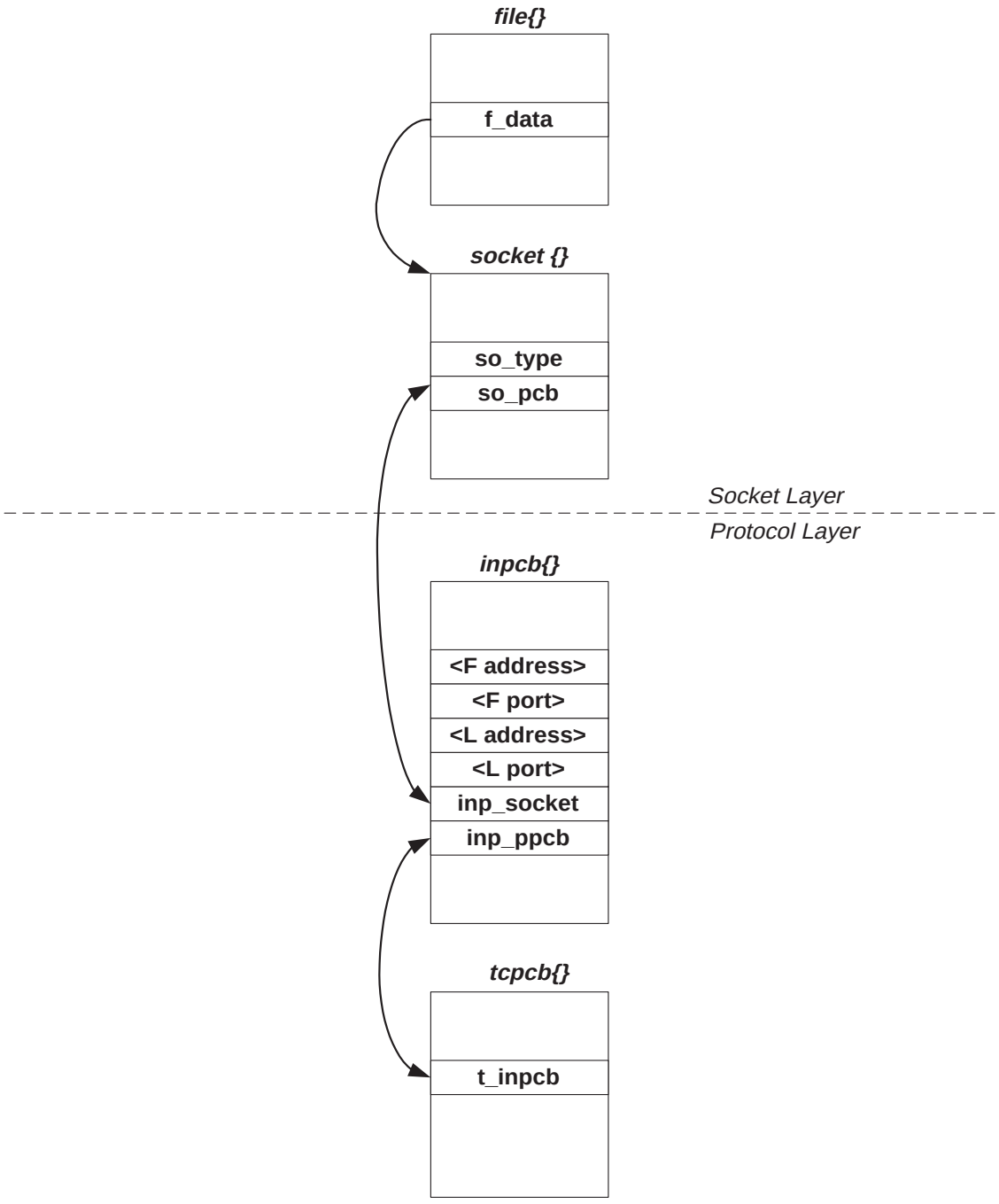

Figure 4.2: Internet protocol control blocks

additions to the TCP control block is much more than that of IP. These changes will be mentioned as we proceed further into the implementation details. There is a one-to-one relationship between the Internet PCB and the TCP control block, and each points to the other. Some of the fields in the structure relevant to our discussion are:

- `tt_rexmt, tt_persist, tt_keep, tt_2msl` and `tt_delack`: These are timers maintained for a particular connection.
- `t_inpcb` : This is the back pointer to the internet PCB.
- `t_state`: This field holds the *state* of the TCP connection.
- `snd_una, snd_nxt, snd_max`: The fields related to the send sequence number space.
- `irs, iss` : These represent the initial receive and send sequence numbers.
- `rcv_nxt, rcv_adv`: Sequence numbers related to the receive sequence number space.
- `snd_wnd, snd_cwnd`: The send window value and the congestion controlled window value. These are used during the exponential slow-start period.

The fields in the structure are better understood after an overview of the TCP state transition diagram, the use of timers in TCP and the way TCP uses the sequence numbers for reliable delivery of data

TCP State Transition Diagram

Many of TCP's actions, in response to different types of segments arriving on a connection, can be summarized in a *State transition diagram* (Figure .4.3). These state transitions define the TCP finite state machine.

The `t_state` member of the control block holds the current state of the TCP endpoint with the values shown below:

1. `CLOSED`:The connection is closed.
2. `LISTEN`: The endpoint is listening for a connection (passive open).

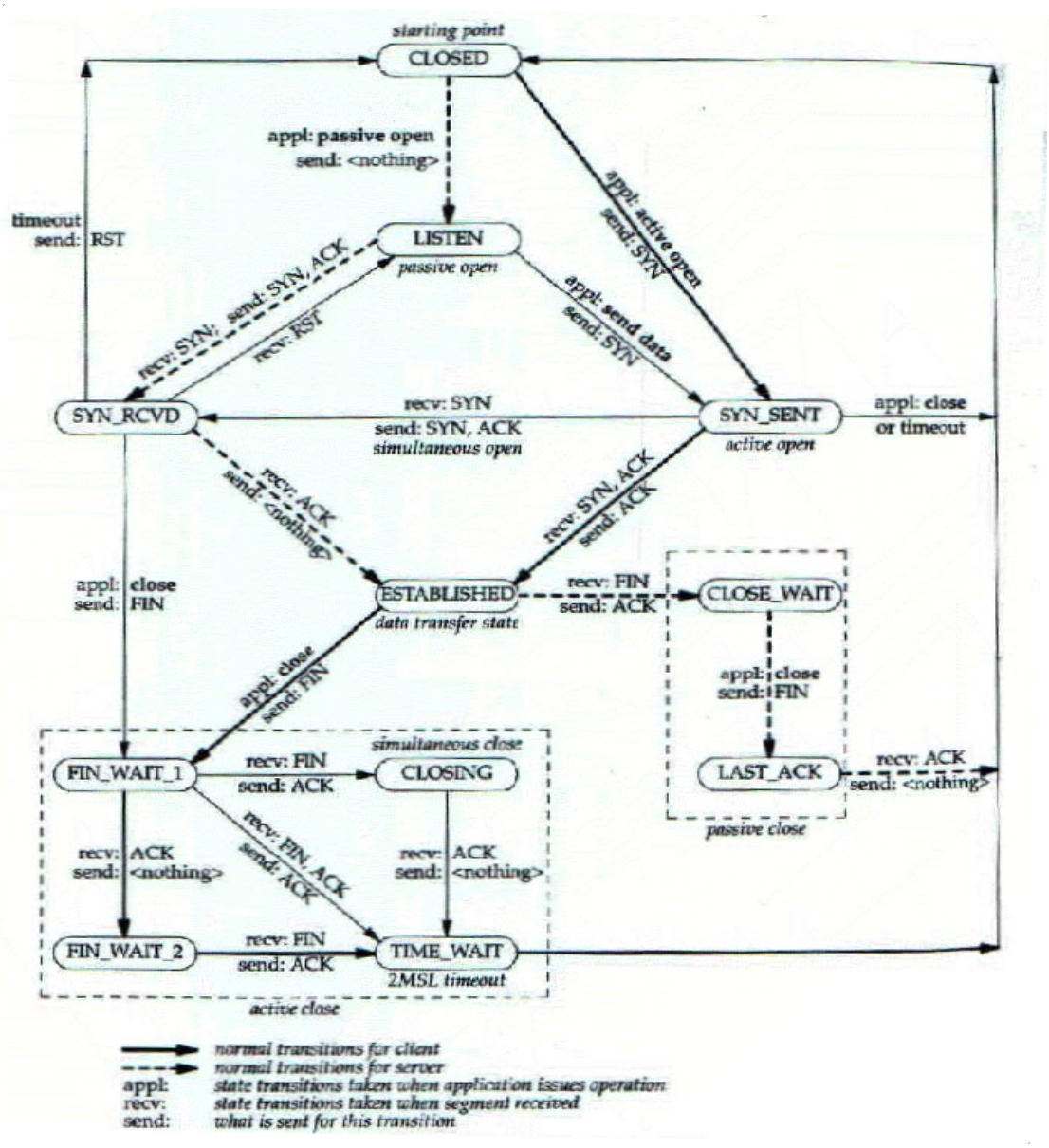

Figure 4.3: TCP state transition diagram

3. SYN_SENT: The endpoint has sent a SYN to its peer (active open).
4. SYN_RECEIVED: The endpoint has sent and received SYN, awaiting ACK.
5. ESTABLISHED: The connection has been established for data transfer.
6. CLOSE_WAIT: The endpoint received FIN from peer, waiting for application to do a close.
7. FIN_WAIT_1: The endpoint has closed the local endpoint, sent FIN to peer, awaiting a FIN and an ACK from peer.
8. CLOSING: A simultaneous close by both the sides, awaiting an ACK from peer.
9. LAST_ACK: The local endpoint has been closed and have also received FIN from the other side, waiting for the final ACK to come from the remote side for the FIN.
10. FIN_WAIT_2: The local endpoint has closed, waiting for the remote endpoint to send the FIN.
11. TIME_WAIT: In this state the connection endpoint stays for 2MSL time (approximately 120s) after doing an active close.

TCP Timers

TCP uses a set of seven logical timers for various purposes. The timer related information is stored in the tt_rexmt, tt_persist, tt_keep, tt_2msl and tt_delack fields of the TCP control block. The following are some of the timers relevant in the implementation of MTCP.

1. A connection-establishment timer starts when a SYN segment is sent to establish a new connection. If a response is not received within 75 seconds, the connection establishment is aborted.
2. A retransmission timer is set when TCP sends data. If the other end does not respond with an ACK to this data when this timer expires, TCP retransmits the data. The value of this timer i.e. the amount of time TCP waits for an acknowledgement is calculated dynamically, based on the RTT (Round Trip Time) measured by TCP for this connection

and also by the number of times this particular segment containing the data is retransmitted. The retransmission timer is bound by TCP to be between 1 and 64 seconds.

3. A delayed ACK timer is set when TCP receives data that must be acknowledged, but need not be acknowledged immediately. Usually in those cases, the ACK will be piggy-backed on the segment containing data from this side. Suppose there is no data sent (and hence no ACK sent) before this timer expires, then a pure ACK is sent to the other side with no data. In other words, this timer represents the maximum time that TCP waits while trying to piggy-back data.

4. A TIME_WAIT timer, often called the 2MSL timer. The term 2MSL means twice the MSL, the maximum segment lifetime of a segment. It is set when the connection enters the TIME_WAIT state and when it expires, the TCP control block and the Internet PCB are deleted, allowing that socket pair to be reused.

TCP Sequence Numbers

Every byte of data exchanged across a TCP connection, along with the SYN and FIN flags, is assigned a 32-bit sequence number. The sequence number field in the TCP header contains the sequence number of the first byte in the segment. The acknowledgement field in the TCP header contains the next sequence number that the sender of the ACK expects to receive, which acknowledges all data bytes through the acknowledgement number minus 1.

Since a TCP connection is full-duplex, each end must maintain a set of sequence numbers for both directions of data flow. In the TCP control block, there are 13 sequence numbers listed: eight for the send direction (the send sequence space) and five for the receive direction (the receive sequence space). The important ones are explained in the following paragraphs.

The following illustrated example (Figure 4.4)shows the relationship between four of the variables in the send sequence space: snd_nxt, snd_una, snd_wnd and snd_max.

An acceptable ACK is one for which the following inequality holds:

snd_una *l* acknowledgement field $\leq$ snd_max

In the above example, an acceptable ACK has an acknowledgement field of 5, 6, or 7. An

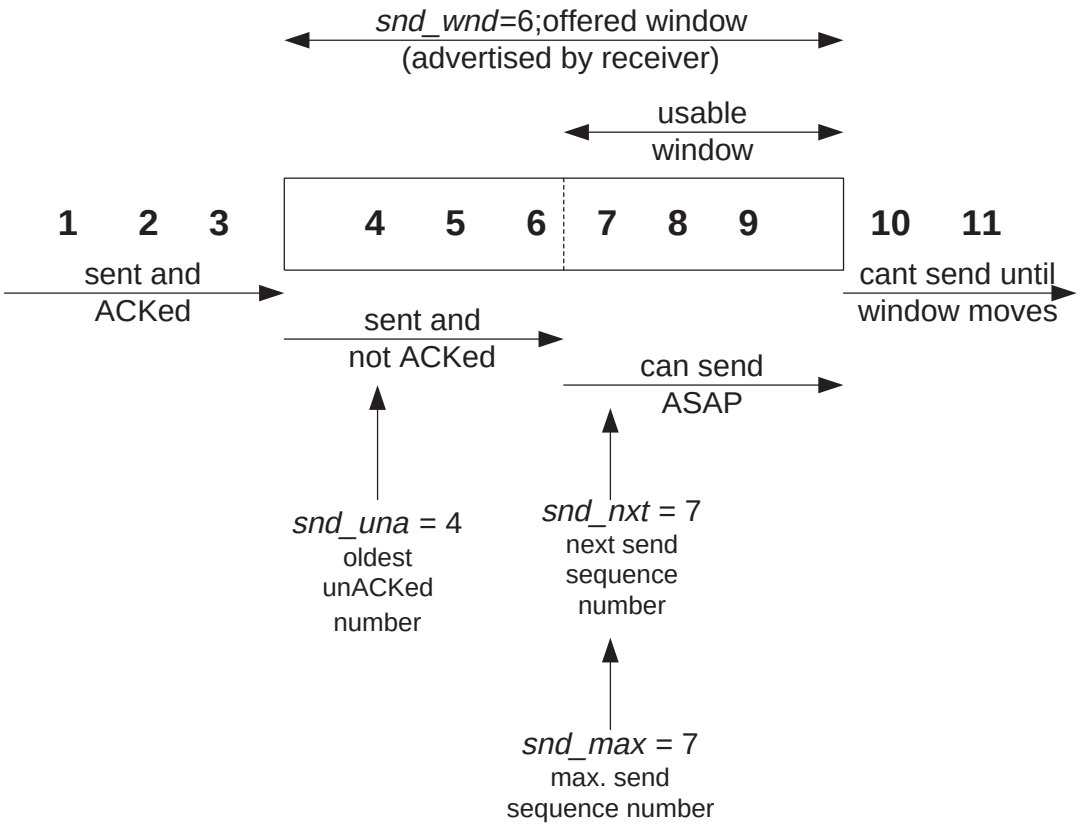

Figure 4.4: Example of send sequence numbers

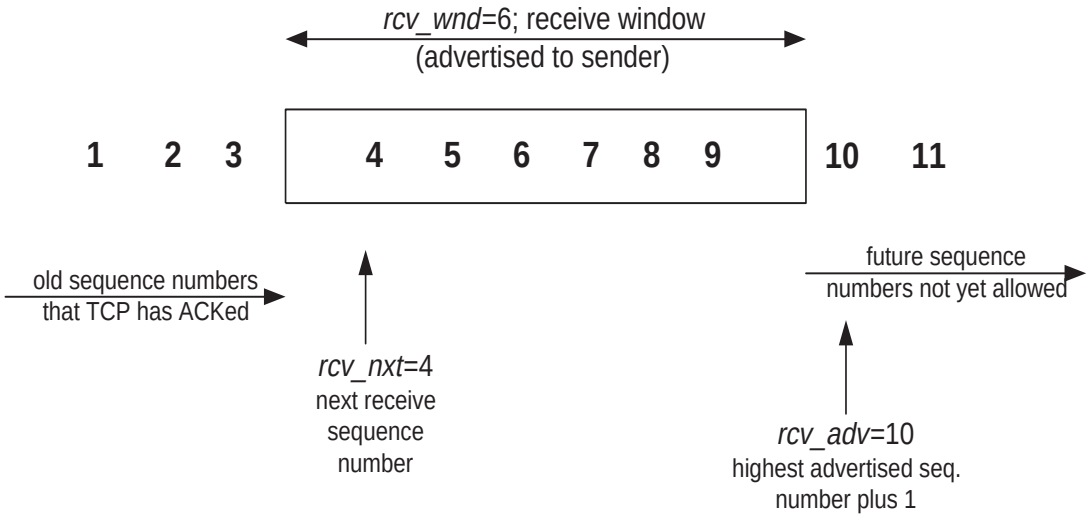

Figure 4.5: Example of receive sequence numbers

acknowledgement field less than or equal to snd_una is a duplicate ACK. If a segment is retransmitted then: snd_nxt $\leq$ snd_max is true.

Figure 4.5 shows the other end of the connection: the receive sequence space, assuming the segment containing the sequence numbers 4,5, and 6 has not yet been received yet. We show three variables rcv_nxt, rcv_wnd and rcv_adv.

The receiver considers a received segment valid if it contains data within the window, that is, either of the following two inequalities are true:

rcv_nxt $<$ beginning sequence number of segment $\leq$ rcv_nxt + rcv_wnd

rcv_nxt $<$ ending sequence number of segment $\leq$ rcv_nxt + rcv_wnd.

TCP Options

The TCP header for a segment can contain options. Every option begins with a 1-byte kind that specifies the type of option followed by the length of the option (1-byte) and the option data itself. The total length of the options field in the header can stretch up to 60 bytes. The implementation of MTCP introduces a number of new options which form the core of the protocol. These options are elaborated upon in the ensuing sections.

4.2 MTCP: The Implementation

4.2.1 API for the Server Application

There are some special system calls that need to be utilized by the server application in order to exploit our mechanism. These system calls are mainly for the kernel to extract application specific information for connection migration purposes.

In the current implementation, all the MTCP specific system calls are built over the existing `setsockopt` and `getsockopt` system calls. In other words, instead of creating new system calls, we have added new socket options for transferring the required information. Semantically speaking, based on the nature of these calls, all of them cannot be categorized as socket options. This approach was mainly taken for convenience. In future we plan to replace these socket options with full-fledged system calls.

The following are the MTCP specific socket options and their function:

1. `TCP_ADDSERVERS`: This socket option was introduced to enable the server application to inform the protocol regarding the location of the ADDitional SERVERS providing the same service. The level of option processing is `IPPROTO_TCP` i.e. at the TCP protocol level. A pointer to `struct extra_ip_addrs` is passed as a parameter and it contains the actual server information. `struct extra_ip_addrs` is a structure introduced specifically for this option. The structure is defined as follows:

```
Struct extra_ip_addrs
{
  connectionID conns[MAX_IP_ADDRS]; /* The optional connection ids*/
  u_int32_t len;           /* The number of ip addresses*/
};
```

   - In the above structure, `MAX_IP_ADDRS` is the maximum number of additional servers that can be provided.
   - `conns` is an array of connection identifiers (the structure connectionID) denoting the additional servers. A connection identifier consists of the tuple <IP address (32

bit unsigned integer), TCP Port number (16 bit unsigned integer)>. Both the IP addresses and the Port numbers are sent in network byte order.

- `len` field represents the actual number of additional servers provided in the structure.

The new kernel data structures and flags that are related to this call are:

(a) An additional option `SO_MIGRATION_SUPPORT` in the `so_options` field of the `socket` structure.

(b) An additional flag `TF_MIGRATE` in the `t_flags` field of the TCP control block (`struct tcpcb`)

(c) An extra field `extra_ips` of `struct extra_ip_addrs` type in the TCP Control Block.

Inside the kernel, this system call does the following three things:

(a) It signals that the corresponding socket has been enabled for Migration Support by setting the bit corresponding to `SO_MIGRATION_SUPPORT` in the `so_options` field of the `socket` structure.

(b) It also sets the `TF_MIGRATE` flag in the `t_flags` field of the corresponding TCP control block. This flag is an indicator to send specific options for this connection in response to appropriate segments.

(c) It copies the information provided by the application i.e. the additional servers to the `extra_ips` field of the corresponding TCP control block.

2. `TCP_CONTROL_MAPPINGS`: The transfer of state between cooperating servers is through an in-kernel connection referred to as the control connection. The network interface through which the server host connects to the clients might be different from the interface used by the server host to communicate with another cooperating server. The cooperating servers might be connected by a high speed interconnect for the transfer of state. This system call provides the mapping information between the two interfaces. The level of socket option processing is the same as in the above case, i.e.`IPPROTO_TCP`. The argument containing

the mapping information is of type `struct ctrl_mappings_info`, which is described below:

```
typedef struct
{
  u_int32_t server_ip_address;      /* the server's ip address that it
                                       * uses to interact with the client */
  u_int32_t control_ip_address; /* Control connection's IP address */
}ctrl_ip_mapping;

#define MAX_CONTROL_IPS MAX_IP_ADDRS
struct ctrl_mappings_info
{
  ctrl_ip_mapping ctrl_mappings[MAX_CONTROL_IPS\];
                                     /* the mappings */
  char num_maps; /* the number of mappings */
};
```

The structure contains an array of type `ctrl_ip_mapping` of length `MAX_CONTROL_IPS` which is same as the maximum number of cooperating servers (`MAX_IP_ADDRS`). The structure `ctrl_ip_mapping` contains nothing but two IP addresses `server_ip_address` and `control_ip_address`, `server_ip_address` corresponds to the interface the server application uses for the clients and the `control_ip_address` corresponds to the interface used for the control connection(s). This mapping is used when a control connection needs to be set up between two cooperating servers for the transfer of state.

A new field `t_control_map_info` of type `struct ctrl_mappings_info` was added to the TCP Control Block structure for storing this mapping data. Inside the kernel, the following actions take place for this system call:

(a) The structure passed from the application-level is copied onto the `t_control_map_info` field of the corresponding TCP control block.

(b) Create a TCP server socket on the designated port `TCP_STATE_SERVICE_PORT` (1001) with an IP address corresponding to the control interface for the given host. For this purpose, the mappings list is searched serially to find the control interface for the host on which the function is called. Then a new TCP socket is created with the socket system call, followed by a call to bind the socket to the IP address representing the control interface. Finally, the listen system call is called on the socket to denote that it is a socket waiting for new connections.

3. `TCP_APP_STATE`: This socket option is used by the server application to transfer(retrieve) application-level state belonging to a particular connection to(from) the kernel. The application-level state is transferred to the kernel by means of the `setsockopt` call while the state is retrieved using the `getsockopt` call. The `getsockopt` call is usually done by the application to retrieve the application-level state of a migrating connection. If this system call returns with a state size of zero, then it indicates that the connection is a new one (no previous snapshots) and not a migrated one.

   The data sent to the kernel via this socket option is an application-defined chunk of memory of a fixed size that represents the application-level state of the connection. The length of this memory chunk (the size of the application state) is sent as the last parameter to the `getsockopt` call. The following actions take place in the kernel for this system call:

   - The chunk of memory passed down from the application is copied to a chain of mbufs and stored in the appropriate TCP control block.
   - The state snaphot is taken. This involves copying of a number of values from the TCP control block like sequence numbers etc. and also certain fields from the socket structure. This system call also achieves the necessary state synchronization. The components of the connection state and the synchronization details would be explained in greater detail in the following sections.

   Another point of observation is that the time at which this system call is made is the appropriate moment to take a *snapshot* of all the state associated with that connection. The application-level state being one of the components of this state *snapshot*. All conforming programs (in concordance with the semantics associated with MTCP) are expected

to make this system call when they have reached a *restart* point in the program. A restart point by definition is such a stage in the program from which the service can resume after migration. Once a migration is complete, the service resumes from the new server from the last restart point provided by the application. For example, the server application might read some data from the socket, do some processing on the data, and might later write to the socket. At this point the server has finished one complete cycle of events on the socket starting with the read, depending on the application, this might be a right time to set the application-state snapshot with the kernel.

4.2.2 Connection State

The connection state snapshot including the application, socket and TCP snapshot are stored in a structure within the TCP control block called `t_conn_state`, which is shown below:

```
struct tcp_conn_state
{
  struct mbuf *app_state;      /* Pointer to the head of the mbufs
                                * representing the app. state */
  int  app_state_size;         /* Size of this app. state */

  tcp_seq rcv_seq_snapshot;    /* The sequence number in the rcv
                                * buffer when the application state
                                * snapshot is taken */
  tcp_seq snd_seq_snapshot ;   /* The sequence number in the snd
                                * buffer when the application state
                                * snapshot is taken */

  struct socket_snapshot so_snapshot; /* The snapshot of the socket */

  /* The snapshot of the tcp endpoint */
  struct tcp_snapshot t_snapshot;
```

```
} t_conn_state;
```

The fields of this structure are:

1. app_state: This is a chain of mbufs storing a chunk of memory representing the application state. This chain is created whenever the application takes a snapshot using the setsockopt call with the option being TCP_APP_STATE.

2. app_state_size: This field represents the size of the application state. In other words, the size of the data stored in the app_state field.

3. rcv_seq_snapshot: This is a sequence number associated with the read buffer (rcv buffer) that represents the point at which the snapshot is taken relative to the incoming stream.

4. snd_seq_snapshot: This is similar to the rcv_seq_snapshot and represents the point at which the snapshot is taken relative to the outgoing stream. This sequence number is computed using the other fields of the TCP control block and also using some state information that is transferred from the previous server. The exact computation is described in section 4.2.3.

5. so_snapshot: This field stores all the state associated with the socket that needs to be transferred for a migration. This field is a structure that is described below:

```
struct socket_snapshot
{
  short so_state;   /*  state of the socket */
  short so_options; /* socket options */
  short so_linger;  /* linger field */
};
```

The first member of this structure is the most significant and it is a copy of the so_state field in the socket structure at the point of snapshot. The so_state field governs the behavior of the socket in its interaction with the application. The so_options and so_linger fields contain values that might be set by the application by various setsockopt

calls. These need to be transferred to the new server to maintain the same socket behavior as was experienced at the old server.

6. `t_snapshot`: This structure captures some of the TCP specific details other than the sequence numbers.

```
struct tcp_snapshot
{
  int t_state;                    /* The state of the TCP endpoint at the
                * time when the snapshot is taken */

  short t_state_transitions;           /* The transitions that need to
                                        * that has taken place since
                                        * the migration to state */

  /* This is to transfer the options */
  u_int32_t t_flags ;         /* The flags of the tcp endpoint */
  u_int32_t t_maxseg;         /* the max segment size set by the user
                                          * as an option */
};
```

The `t_state` field is the TCP state of the connection at the time of the snapshot.

The `t_state_transitions` is a log of all the state transitions that have taken place since the last snapshot. State snapshots can be taken in all the TCP states where the application has access to the descriptor for the TCP connection. The application has access to the descriptor as long as the application has not called ”close” on the socket. Consider a scenario, in which the application took a snapshot when the connection is in the ESTABLISHED state, later the connection makes a transition to some other state that can be reached from the ESTABLISHED state. Suppose, a decision to migrate is taken now; the new server would resume from the ESTABLISHED state. If the state transitions are not logged then the kernel at the new server is unaware of any of the state transitions made

by the connection at the old server after the snapshot. If the old server had attempted to close the connection first (an active close) then semantically even the new server has to replicate this action by doing an active close. In the scenario being described as the new server has resumed from the state of ESTABLISHED, it might be possible that the new server might receive a FIN from the client (an acknowledged passive close from the client) as a response to the active close of the old server. The new server might be discarding duplicate writes by the application when this FIN is received i.e. a FIN when the connection endpoint is in the ESTABLISHED state. The new server connection endpoint would prepare itself for a passive close even though the original connection endpoint had done an active close. *This leads to a situation where both the endpoints of the connection are attempting a passive which results in a deadlock.* This explains that a log of all the state transitions since the last snapshot. All the state transitions present in the log have to be replayed in the same order at the new server before accepting any FINs from the client.

The remaining two fields `t_flags` and `t_maxseg` are a copy of the `t_flags` and `t_maxseg` fields in the TCP control block at the time of the snapshot. These hold values that might have been modified by previous `setsockopt` (with the level set as TCP) calls of the server application.

The various fields in the `t_conn_state` structure are set when the application takes a snapshot using the `TCP_APP_STATE setsockopt` call. A few other components of the state are stored in the socket structure rather than in the TCP control block mainly for performance reasons. The following are the state components that are enclosed in the `socket` structure:

1. `read_log_mb`: This is a chain of `mbuf`s that stores all the complete `mbuf`s read from the buffers since the last snapshot. This chain of `mbuf`s needs to be transferred to the new server in the event of a migration.

2. `so_read_log_len`: The size of the data in the `read_log_mb` mbuf chain.

3. `so_read_log_offset`: The data in the first `mbuf` present in the read buffer that is part of the log. So the actual length of the log is the size of data in the mbuf chain i.e.

so_read_log_len plus this offset. We had to split the logged data buffers into two parts, i.e. complete mbufs + a part of the first mbuf in the read buffer, primarily for efficiency reasons. We employ a zero copy method to maintain this log. When data is read by the application from the read buffer and if it consumes an entire mbuf then the mbuf is normally freed. In our case, we do not free this mbuf but append to the so_read_log_mb chain. In case the data read by the application did not cross one complete mbuf then m_data field in the relevant mbuf is advanced by the amount of data read. To log accurately the partially read mbuf we keep the amount of partially read data in the so_read_log_offset field.

4.2.3 Steps in a Typical Connection Migration

This section describes in detail all the actions that take place for a typical migration of a connection. The details regarding the entity that triggers the connection migration and under what circumstances it does so are orthogonal to this discussion.

The preparation for a potential connection migration starts when the client does the connection session establishment with the first server. This is illustrated in the Figure 4.6. Figure 4.6 describes the connection establishment between the client and the origin server belonging to a service provided by a group of cooperating servers. The origin server, destination server and the client are denoted by S1, S2 and C respectively. The origin server is the server belonging to a *service pool* that the client contacts first.

The client C attempts to establish a connection by sending a SYN to S1, if S1 is able to accommodate this client then it replies with a SYN+ACK to C. In addition to the SYN+ACK the server sends a TCP option TCP_MIGRATE with the option data being the set of alternate servers and ports describing the other cooperating servers of the service. At S1, this information is obtained from the extra_ip_addrs field of the corresponding TCP control block of this nascent connection. For the extra_ip_addrs field of the control block to have this information, it should be populated by a previous setsockopt-TCP_ADD_SERVERS system call from the application. On receiving the TCP option, the client copies the alternate servers information to its extra_ip_addrs field. To distinguish the nature and use of the same field at the two different ends of the connection, each end is tagged with a different flag in the

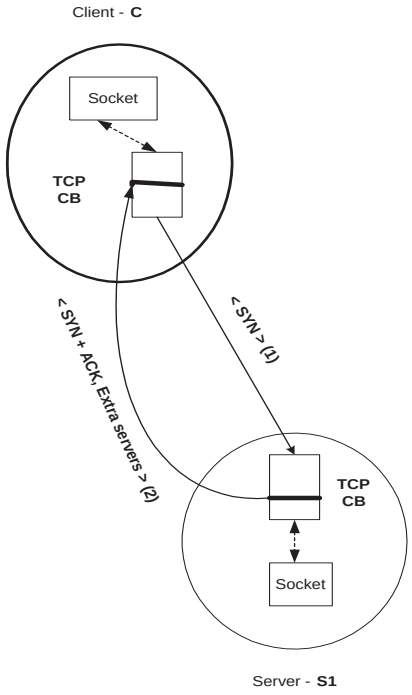

Figure 4.6: Additional servers information transfer via TCP option

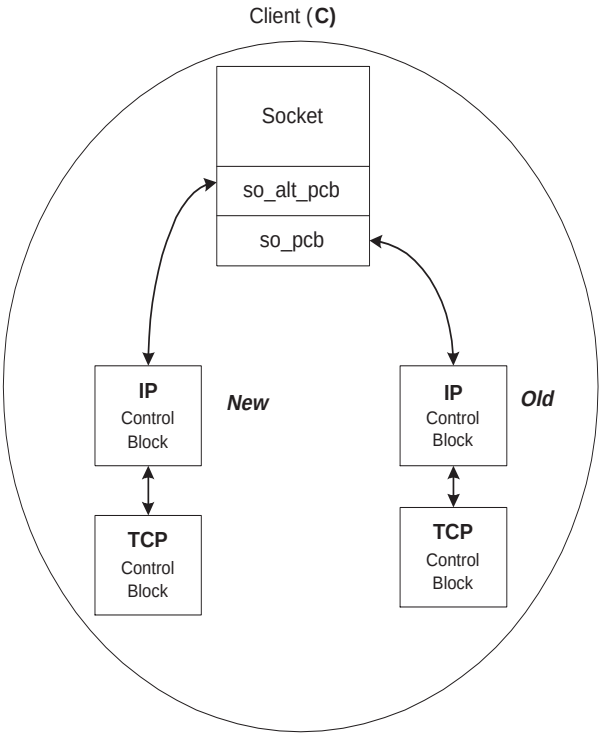

Figure 4.7: Socket structure at the client side.

so_options field of the socket structure. At the server endpoint the so_options field has the flag SO_MIGRATION_SUPPORT set implying that the connection endpoint has support for migration. At the client endpoint, so_options field has the SO_CAN_MIGRATE flag set implying that this end of the service session can migrate to different servers during the course of the session.

The protocol at the client side chooses the next server in the list of alternate servers provided by S1 and initiates a new connection with the second server (hereafter S2) by sending a special SYN. The connection initiation with the new server is done at the level of the transport layer protocol. Thus new TCP and IP control blocks are created for the connection but the same socket structure is retained (Figure 4.7). To accommodate the new connection in the same socket the so_alt_pcb field in the socket points to the new IP control block, whereas the so_pcb field points to the IP control block of the connection with S1. As operations from the application on the socket get routed through the so_pcb field, the new connection is shielded from such operations when it is not yet established. However, both the control blocks have

pointers to the socket structure, thus operations from the network below of both connections can access the socket structure. To denote an ongoing migration, the socket state is changed to SS_ISMIGRATING. The impact of this state change on the packets entering or leaving the network will be explained shortly. The SYN sent to S2 contains the TCP option TCP_MIGRATED_CONN with the option data containing the following fields:

1. The connection identifier of the connection between C and S1 i.e. the ip addresses and port numbers of both endpoints.

2. The value of the rcv_nxt field in the client TCP control block of the original connection. This value is needed for synchronization purposes and its exact use is explained when we explain the actions that take place at S1 for a connection migration.

At the new server S2, the TCP_MIGRATED_CONN option discriminates between a migrated connection and a new connection. The actions that follow are illustrated in Figure 4.8. When the server application does a TCP_CONTROL_MAPPINGS-setsockopt call, it starts a in-kernel listening server socket binding on the port 1001 (STATE_PORT). This listening socket acts as a server for transferring state information of all the TCP connections in that particular host when requested by other interested peers.

The interface to which this server binds might be the same interface as the one used by the host for other clients (i.e. service clients) or it could be a separate interface (termed hereafter as the *control* interface). This interface is provided through an argument in the TCP_CONTROL_MAPPINGS system call. This connection between a pair cooperating servers which is used exclusively for the transfer of state information is denoted as the control connection. There exists exactly one control connection between any two cooperating servers and the transfer of the state information of all the migrating connections are multiplexed onto this connection. To get efficient access to the control connection, a pointer to the socket structure of the control connections is stored in a hash table hashed by the IP address of the control interface of the cooperating peer.

When S2 requires the state information of the connection from S1, the following steps take place:

1. It gets the service IP address of the S1 from the TCP_MIGRATED_CONN option sent along

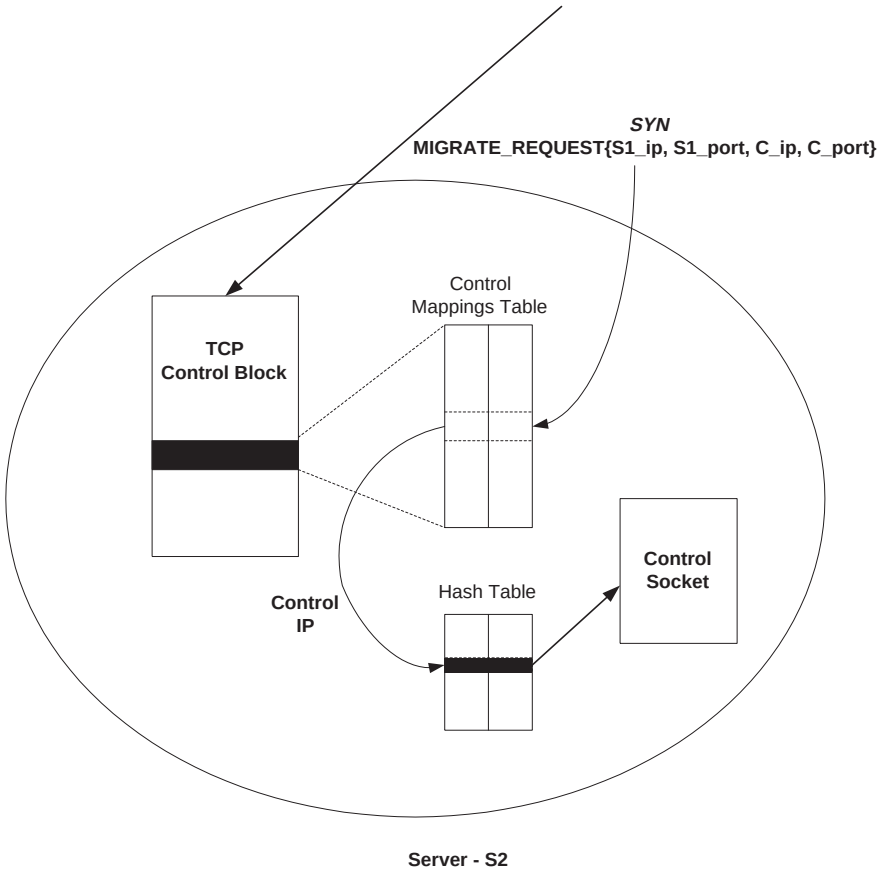

Figure 4.8: Actions at server S2 on arrival of a migration request

with the SYN to S2. The service IP address is the address the server application at S1 binds to in order to serve the interested clients. This may be different from the IP address used by S1 for the control connection.

2. It searches for the appropriate IP address of S1's control interface (control IP address) from the `CONTROL_MAPPINGS` structure stored in the TCP control block.

3. A lookup into the control connections hash table is done using the control IP address of S1, and the pointer to the socket structure of the control connection is retrieved. If there is no socket structure then a new control connection is created to S1 and an entry is made in the hash table when the TCP connection is completed.

4. A request for the state is sent on the control connection. The sequence of events following this step cannot be presented without an understanding of the **state request** and **state reply** message details sent via the *control connection*. We digress here to discuss how this transfer of state takes place on the *control connection*.

Control Connection and State Transfer Details

The **state request** message has the following fields:

1. The four-tuple *connection id* of the connection that is migrating.

2. The four_tuple *connection id* of the new server endpoint that is being created for this migrating connection. When the state has been transferred to the new server host by means of a **state reply** message, this field acts as a key to identify the control block that requested the state.

3. The `rcv_nxt` field sent by the client in the TCP option. This field is used to optimize the amount of state transferred.

Once S1 receives the **state request** message, it starts preparing and packaging the state information to send it to S2. The state information is packaged in the **state reply** message. In case the *connection id* present in the **state request** message does not correspond to a valid connection in the host then a **state reject** message is sent back. The following are the sequence of events that take place while packing the state information at S1:

1. A pointer to the *key* in the **state request** message identifying the requesting connection is maintained. This is needed because we need to copy the key over to the **state reply** message.

2. A hashtable lookup on the TCP control blocks (representing all the valid connections) using the *connection id* of the migrating connection is performed.

3. In case of an unsuccessful lookup, a **state reject** message is sent straight away. A successful lookup implies that there exists a connection endpoint at the given host. However, it does not imply that this endpoint is valid for a migration. This is so because in certain cases existence of a TCP control block doesn't mean that the connection still has data to be transferred. Unless there is a case for further data transfer on a particular connection, migration is useless. The TCP state stored in the TCP control block is an indicator of subsequent data transfers on a connection. In the TCP states of `SYN_SENT, SYN_RECEIVED` i.e. states before the state changes to `ESTABLISHED` no data has been exchanged at all, and hence a migration of the connection is wasteful i.e. state request for such a nascent connection is invalid. Similarly, the states `LAST_ACK` and `TIME_WAIT` imply that all data have been transferred and hence migrations in these cases are also invalid. Cases where the TCP connection has sent/received all data but is waiting for a FIN are valid because semantically FIN is part of the sequence number space and hence for all practical purposes a part of the data. In all the cases where the connection migration is invalid, a **state reject** message is sent back. The **state reject** message has the following format:

   (a) An integer representing the reason for the reject (values are `CB_NOT_FOUND`, `INVALID_STATE`).

   (b) The key representing the requesting connection.

4. The packing of the three core components of the state information is done at the server.

   The most straight forward of these is the **application-level state** component. This data is picked directly from the corresponding fields in the TCP control block. The length and the application state are in the `t_conn_state` structure within the TCP control block as `app_state_length` and `app_state` respectively.

The calculation of the **write buffer state** length involves the use of many sequence number related values like snd_una stored in the TCP control block of the original connection endpoint at S1. The calculation of the state length (henceforth referred to as snd_state_length) is done as follows:

(a) The value of the field rcv_nxt from the client sent in the **state request** message at the time of migration is used for pruning the size of the write buffer state. It is possible that some of the ACKs sent by the client for the data of S1 might not have reached S1. Thus the rcv_nxt value at the client might be greater than the snd_una value present at S1. If the rcv_nxt value is indeed greater than the snd_una perceived by S1, then the difference (rcv_nxt - snd_una) can be discarded from the buffer and might lead to a decrese in the state data to be transferred.

(b) The maximum of the two above mentioned values is computed and stored in a variable called snd_max_seen = max (rcv_nxt of client, snd_una at S1). This value is significant and is used further in other messages during the migration to ensure consistency.

(c) During the calculation of the snd_state_length the number of duplicate bytes to be discarded at the new server is also computed (this value is referred to as snd_bytes_discard).

(d) If snd_max_seen is less than (snd_seq_snaphot value in the TCP control block) then the snd_state_length is equal to (snd_seq_snapshot - snd_max_seen) and the value of snd_bytes_discard is 0, else snd_bytes_discard is equal to (snd_max_seen - snd_seq_snapshot) and the snd_state_length is zero. Both scenarios are described in the Figure 4.9. The first figure is self explanatory, the rcv_nxt sent from the client side of the connection acts as an implicit ACK via S2. In the second figure, the client has received upto snd_max_seen and the snapshot falls before this value. So when the new server starts sending data (logically) starting from the snd_seq_snapshot sequence number it would be sending duplicate data till the snd_max_seen sequence number (shown shaded in the figure). One point to observe is that the write buffer state data is readily available in the write buffer

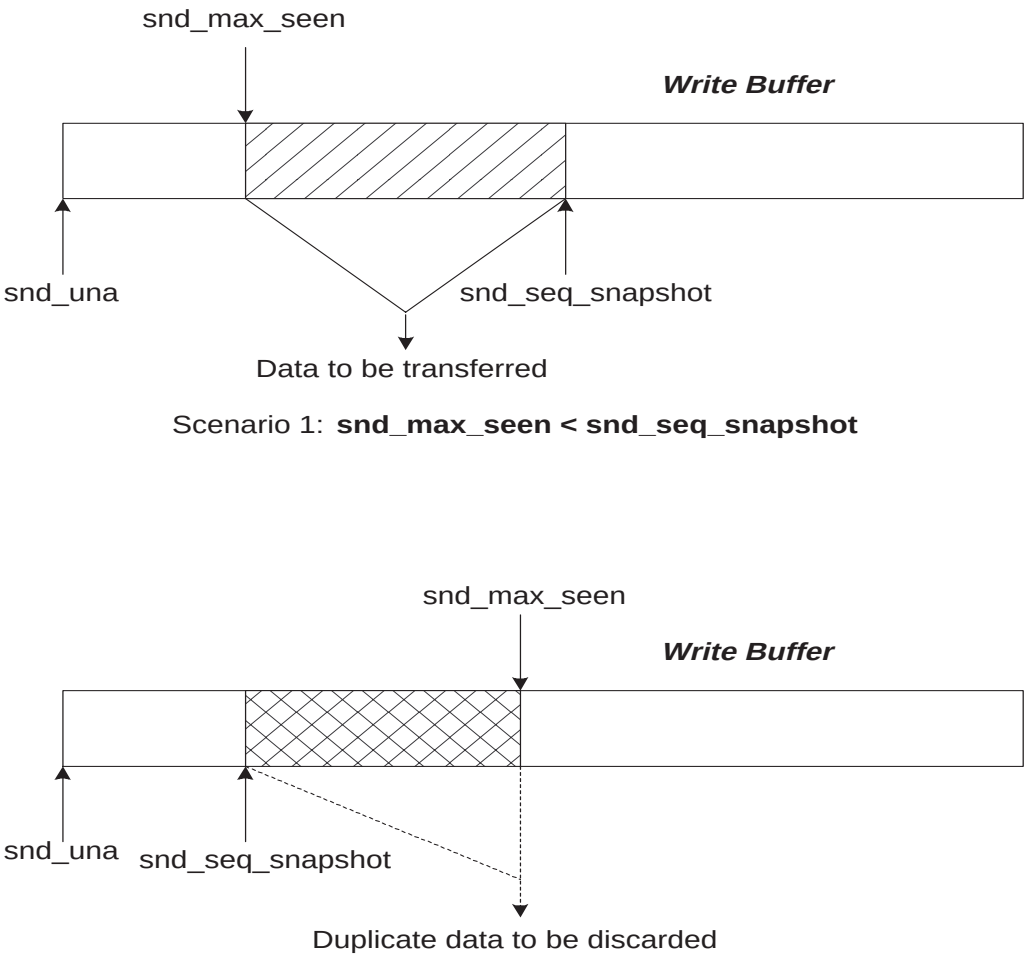

Figure 4.9: Relation between `snd_max_seen` and `snd_seq_snapshot`.

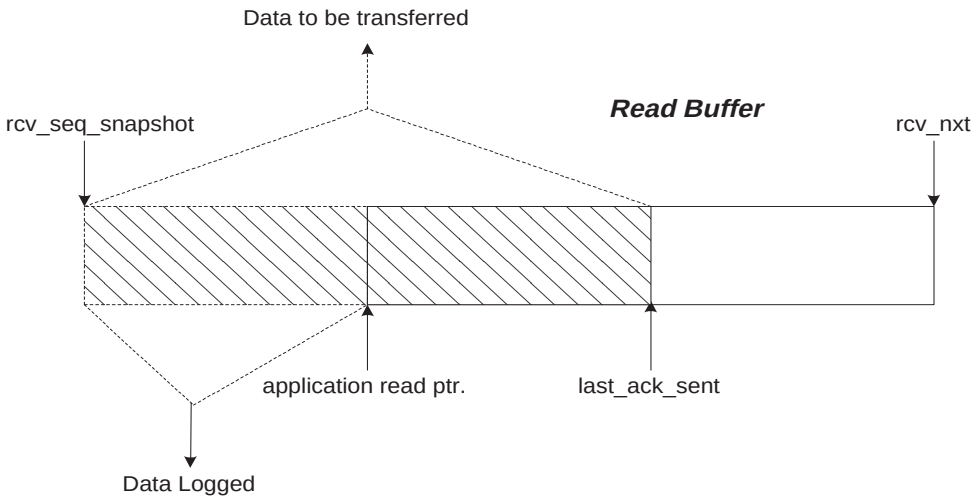

Figure 4.10: Read buffer synchronization implementation details.

corresponding to the connection.

The state data related to the **read buffer** is described next. The computation of its length (`rcv_state_length`) is explained below:

(a) The `rcv_state_length` is (`last_ack_sent - rcv_seq_snapshot`) and is zero if this difference is negative. Both variables correspond to the TCP control block of the original connection endpoint at S1.

(b) However, there is one significant difference compared to the write buffer case. The `rcv_seq_snapshot` might be behind in sequence number terms compared to the smallest sequence number stored in the read buffer. This scenario is depicted in Figure 4.10. The shaded region shows the data that is part of the TCP connection state. The read buffer contains only the data between the application read pointer and `rcv_nxt`. The remaining part of the state is furnished from the read buffer data that we have logged since the last snapshot in the `socket` structure.

5. Once the three component lengths have been computed, the header for the **state reply** message is created. The header of the **state reply** message contains the following fields.

(a) A tag indicating that it is a **state reply** message for a particular state request.

(b) The key for identifying the requesting connection endpoint i.e. the *connection id* of the new server's endpoint.

(c) The application state length (`app_state_length`).

(d) The write buffer state length (`snd_state_length`).

(e) The read buffer state length (`rcv_state_length`).

(f) The greater of the two values : `rcv_seq_snapshot` and `last_ack_sent` of the original connection endpoint. This field has the same effect as sending the `rcv_nxt` from the client to S1 via S2 to act as an *implicit* ACK.

(g) The number of duplicate bytes to be discarded (`snd_bytes_discard`).

(h) The `snd_max_seen` value computed earlier. The significance of sending this field will be clear in a later section of this discussion.

(i) The `t_snapshot` field in the `t_conn_state` structure embedded in the TCP control block.

(j) The `so_snapshot` field in the `t_conn_state` structure contained in the TCP control block.

The header is followed by the actual data belonging to the three components. The entire state information i.e. header plus data is organized as a chain of `mbufs` and is sent over the same control connection on which the **state request** was received.

Once the **state reply** reaches S2, the following events take place in sequence:

1. The key from the **state reply** message is extracted and a lookup for the appropriate TCP control block is made. The lookup on the control block might be unsuccessful because of timing issues. When a new connection endpoint is set up for a SYN received from a peer, a *connection establishment* timer is started. The duration of the timer is 75 seconds. If the peer to which the SYN was sent doesn't acknowledge your SYN before this timer expires, then the TCP control block is dropped from the queue of connections. Thus if the time taken by S1 to pack and send the **state reply** exceeds this limit then the connection is dropped.

2. On a successful lookup, the lengths of the three core components in the header of the **state reply** message are retrieved. In addition, the number of duplicate bytes to be discarded in the S2 to C connection (`snd_bytes_discard` calculated in the preceding paragraph) is retrieved and stored in the `so_snd_bytes_dups` field of the new server endpoint's `socket`. The `snd_max_seen` is also retrieved from the message header and stored in the `orig_server_max_snd_seq` field of the `orig_connection_info` structure enclosed in the TCP control block of the new server endpoint. The use of this value is explained in a following paragraph. Similarly, the origin server's (S1's) `last_ack_sent` sent in the message header is copied onto the `orig_server_rcv_nxt` field of the `orig_connection_info` structure. This value is copied to serve as an *implicit* ACK via S2 for the data sent by the client to S1.

3. The actual data representing the application state, the read buffer and the write buffer states are extracted and appended to the appropriate buffers in the new endpoint. The log representing any state changes after the snapshot is also copied onto the local TCP control block.

4. A **state ack** message implying that the **state reply** message was received and processed is sent to S1. Up until this point, the connection endpoint at S1 behaves exactly like a normal connection endpoint and does nothing as a result of transferring the state information to S2. The **state ack** message would contain the *connection id* for the migrated connection.

5. Now, the new server endpoint has all the information necessary to accommodate the client and resume service to it. It signals the acceptance of the client according to the normal semantics of TCP by sending a SYN+ACK on the connection.

S1 on receiving the **state ack** reply for the migrated connection, initiates a TCP *passive close* on the migrated connection and puts the endpoint to `CLOSED` state. This completes a typical interaction between two cooperating servers over the control connection for the transfer of state of a migrating connection.

While sending the SYN+ACK back to the client, S2 in addition sends an extra option that contains the following two values from the TCP control block:

1. The origin server (S1) endpoint's `rcv_nxt`.

2. The `snd_max_seen` value.

The action now shifts to the client endpoint of the S2-C connection. One point to observe is that the client endpoint's `socket` is still in the `MIGRATING` state and all the actions on the socket are governed by this. In this state, no new data is sent to S1 on the old connection, as it does not make sense to keep sending data once you have started migration. However, data from S1 is accepted as usual. The client's data that is ACKed by S1 is not deleted from the buffers in this state. Similarly, S1 doesn't send any ACKs for the data received from the client after the migration of state.

The main reason for buffering at the client's side is because the client is unaware of the exact point at which the transfer of state from S1 happens after the initiation of migration. Consider the following scenario to elucidate the need for this buffering. In Figure 4.11, `delta` represents the difference between `last_ack_sent` at S1 and the client's `snd_una` at the moment the state request message is processed at S1. This represents the ACKs sent by S1 that are still in the network when the packing of connection state is done at S1. The state transferred to S2 is made with reference to `last_ack_sent`. Before the completion of migration i.e. when the **state ACK message** is not yet received for the **state reply** message of S1, lets say S1 sends an ACK of length 'K' after the transfer of state as shown in the Figure 4.12. In the meantime, lets assume that the client has received the ACK for the `delta` so `snd_una` at the client's side shifts to the right by `delta`. Now, if the ACK of length 'K' is accepted at the client and discarded before S2 starts resuming the service, then this data (of length 'K') never reaches S2 and is lost forever (as it is neither at the client nor at the server). Thus for maintaining the reliable data delivery semantics of TCP we need to prevent the client from accepting any ACKs from S1 after starting the migration. In this scenario, the ACK for `delta` never reaches the client. The way the client makes up for the lost ACKs is through the *implicit* ACK explained in the next paragraph.

The following are the events that take place in sequence at the client endpoint when the SYN+ACK is received from S2.

1. The switch from the old server to the new server is now completed. The old endpoint's IP

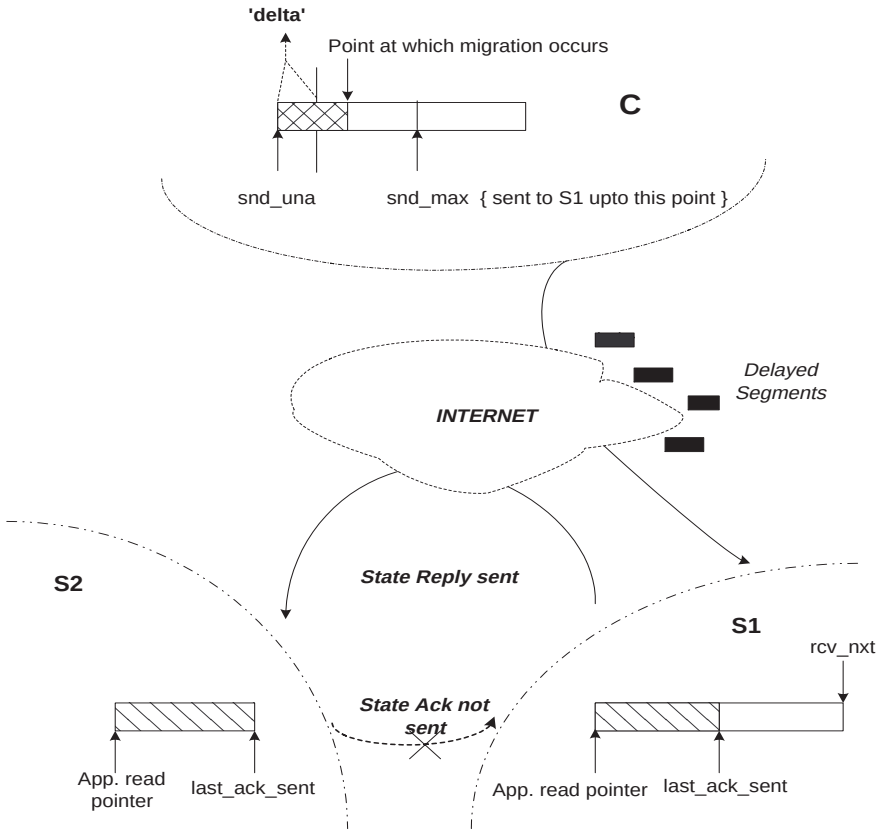

Figure 4.11: Client side logging to maintain exactly-once semantics - I.

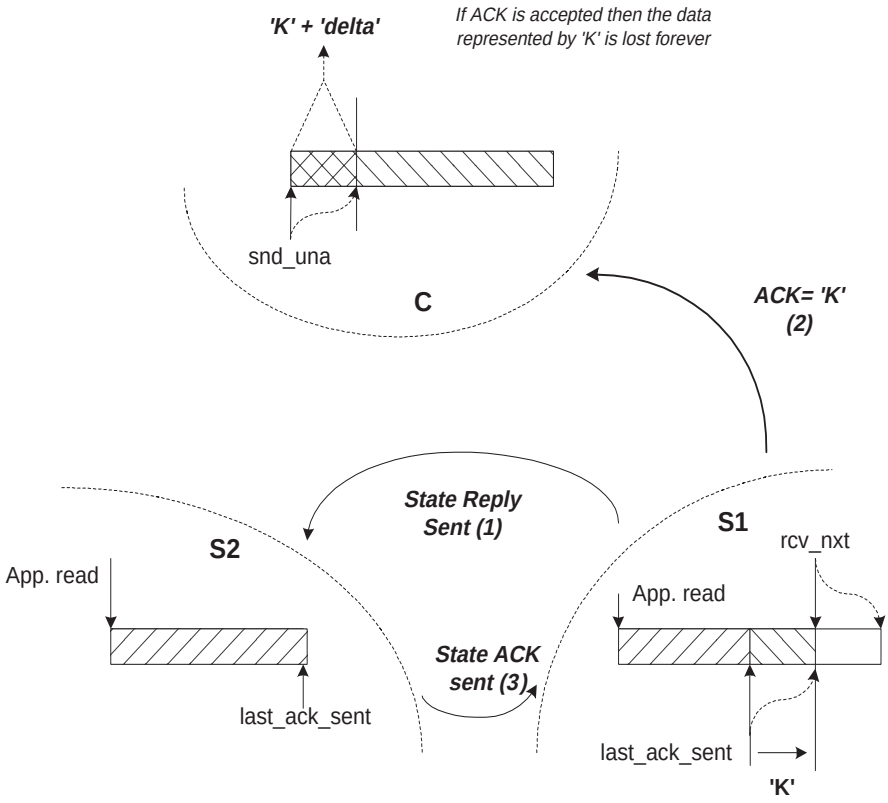

Figure 4.12: Client side logging to maintain exactly-once semantics - II.

control block is now shifted to so_alt_pcb and the new endpoint (the one representing the connection to S2) is shifted to *so_pcb*. This signifies that all new data from above the socket would be routed to the new server and not to the old server.

2. Certain relevant information from the old connection endpoint's TCP control block is copied to the new connection endpoint's TCP control block. The copied data includes the following:

   (a) The t_flags field.

   (b) The extra_ips structure.

   (c) The maximum segment size (t_maxseg field).

   (d) The rcv_nxt field in the old TCP control block is copied onto the t_last_rcv_nxt field on the new TCP control block. The significance in storing this value will be explained shortly.

3. The difference between the snd_una of the old endpoint and the last_ack_sent sent by S1 via S2 through the TCP option is calculated. In case of a non-zero difference, the send buffer is pruned in the front by this difference. This is another implicit ACK analogous to the one on the server side. In case this implicit ACK covers a FIN sent by the client then we need to take appropriate actions at the client side like a change in the state of the TCP endpoint. This *implicit ACK* makes up for the ACKs rejected by the client endpoint in the SS_ISMIGRATING state.

4. The TCP control block representing the old connection endpoint is put in the **BLACKHOLE** state. This state that we have introduced is closest in its operation to the TIME_WAIT state. It is similar to the TIME_WAIT state in its behavior of waiting for a time of 2MSL before moving to the CLOSED state. It differs from the TIME_WAIT state when it receives any data from its peer. In the TIME_WAIT state the endpoint will respond with an ACK, whereas the endpoint in the **BLACKHOLE** state doesn't send anything back.

5. Finally, an ACK is sent by the client in response to S2's SYN. As the client is continuously receiving new data from S1 when the migration is in progress, the current largest sequence number in the read buffer (rcv_nxt) might have overshot the snd_max_seen

calculated at S1. However, the buffer at S2 is consistent with `snd_max_seen` and not with the new value of `rcv_nxt`. The difference (termed as β) between `snd_max_seen` (obtained by the client via an option on the SYN+ACK from S2) and `rcv_nxt` (captured in the `t_last_rcv_nxt` field above) is sent as an option piggybacked in this ACK. Sending this value is critical to maintain the exactly once semantics of TCP.

On receiving the ACK for the SYN at S2, the migration is complete at the server endpoint too. It extracts the β from the TCP option. If there is some data in the write buffer transferred from S1, the data of length β from the beginning of the buffer is discarded. In case there is no data, then this value is added to the `so_snd_bytes_dups` value present in the socket increasing the amount of duplicate data to be discarded. Certain state information obtained from S1, that couldn't be reinstated before (i.e. during the connection establishment stage) like the TCP state and the socket state are reinstated to the new connection endpoint now. This also includes the TCP options and the other socket options. If the socket contains any outstanding data to be sent, then this data is pushed out immediately. One key point that we have previously glossed over deliberately because of lack of proper context is the method of computing the sequence number values at snapshot: `snd_seq_snaphot` and `rcv_seq_snapshot`. They are computed as follows:

`snd_seq_snapshot` = `snd_una` - `so_snd_bytes_dups` + `so_snd.sb_cc`.

`rcv_seq_snapshot` = `rcv_nxt` - `so_rcv.sb_cc`.

`so_snd.sb_cc` and `so_rcv.sb_cc` represents the total size of data present in the write and read buffers respectively.

This completes one typical connection migration from one server to another cooperating peer server. The exact time for the migration depends on a number of factors. The primary among those are:

- *Characteristics of the path* between the client and the new server. The delay/bandwidth characteristics along this path determines whether the migration is feasible or not.

- The type of *network connectivity between the two cooperating servers*. The time to migrate a connection depends heavily on the time to transfer the state information between the participating servers in a *lazy* connection migration mechanism. A dedicated high

bandwidth and/or low delay connectivity between the two servers might lead to a big difference in terms of response time perceived at the client.

- The *load on the old server* at the time of servicing the connection state request from the new server. This is a shortcoming of having a lazy connection migration mechanism. The old server might not even process the request for the connection state from the new server before the connection establishment timer expires at the new server. In this case, the connection remains with the old server and the resultant service is poor to the client.

- The *policies* employed to make the migration decisions. A good policy to trigger a migration might reflect in a lesser time for migration. For example, a good policy decision could anticipate a loaded server earlier and facilitate in transferring state information when the old server is comparatively lightly loaded.

Chapter 5

Performance Evaluation

In this chapter, we describe the experimental setup followed by the motivation and methodology behind the experiments. We then present the results of our experiments.

5.1 Experimental Setup

The experimental test-bed of three workstation class machines all of the same configuration: Intel Celeron 450MHz processor, 256MB RAM. All the machines were connected by a 100Mbps Ethernet link, and were isolated from outside network traffic by means of a dedicated hub. All the three machines were loaded with a modified FreeBSD kernel containing the implementation of MTCP. Two of the machines played the role of two cooperating servers providing the same service, and the third one played the role of a client to these server applications. This setup is illustrated in Figure 5.1.

In addition to the 100Mbps Ethernet interface, the server machines are connected by an exclusive link by means of an extra 100Mbps Ethernet interface dedicated for the purpose of transferring connection state between them.

5.2 Goals

Only a preliminary performance evaluation is described in this and the forthcoming paragraphs. Experimental evaluation of our implementation is not complete with these experiments and is still in progress. The experiments were performed with a view to check the feasibility of the mechanism and the performance of the implementation under different scenarios.

The primary goals of the experiments performed are as described.

1. The first experiment was a micro-benchmark conducted to get a measure of the times

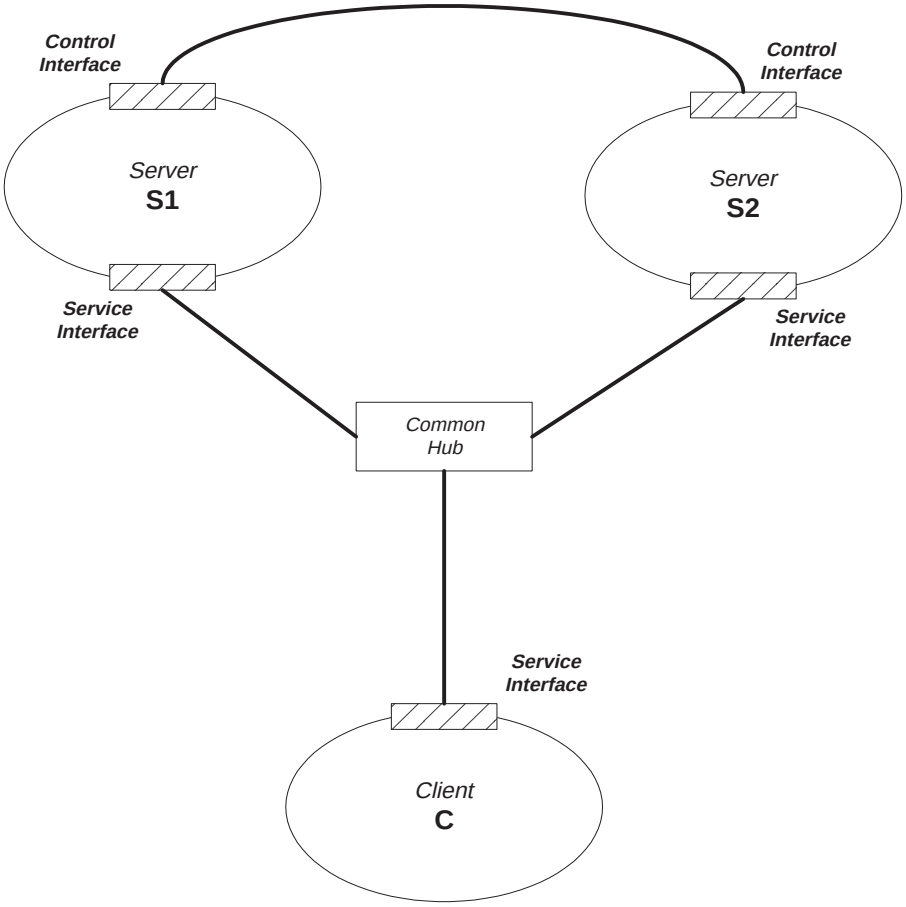

Figure 5.1: Experimental test-bed

taken by the different components in a migration and their relative weights.

2. The second experiment was designed as a macro-benchmark to see how complete migration-aware applications (client and server) would perform over a network that changes its characteristics dynamically.

5.3 Methodology and Preliminary Results

This section deals with a more detailed description of the experiments followed by the preliminary results obtained in those experiments.

5.3.1 Experiment I: *Microbenchmark*

This experiment was designed to find the relative contributions of the various steps in a complete connection migration to the total migration time and their dependency on the state size. The experiment was also designed to validate some of the decisions taken during the implementation regarding some data structures and algorithms, as this would be reflected in the values obtained for the total time for migration.

For the experiment, we employed a migration-aware synthesized server and client applications. The server application was designed such that at any point of time the state to be transferred from an old server to a new one would consist of only *the application state component* i.e. the buffered data in the read and write buffers of the connection would not be reflected in the state at all. This was done primarily to have the same size for the connection state information across different iterations of the program. The application state component's size can be controlled at the application level at a proper value. The components related to the buffer change dynamically and the size of which is highly dependent on the time at which the *state request* message is processed at the old server. This quantity cannot be maintained constant across different iterations of the same experiment.

The experiment was conducted with different *connection state information* sizes (which is the sum of the application level state size and the constant headers used for the messages). Each run for a particular state size was averaged over 200 iterations. Each iteration consisted of a single one-way connection migration of the connection from server S1 to server S2 (Figure

State size	Client			Server 2			Server 1
	T*prep_migr*	T*wait_migr*	T*complete*	T*prep_sr*	T*wait_sr*	T*reinstate_s*	T*prep_s*
0	73	490	10	40	191	20	50
5K	87	1002	7	42	784	25	89
10K	90	1621	7	42	1382	27	120

Figure 5.2: Time spent in various stages of connection migration at client, origin and destination servers for different state sizes

3.1).

The table in Fig. 5.2 shows the times for the different steps in a typical migration for varying connection state sizes. The different steps measured and their dependency on the state size are as follows:

- **Client side**
  - *Tprep_migr*: The time taken at the client to prepare the SYN to Server 2. The time is for retrieving the connection identifier of the client's connection to server S1 and creating a TCP option for sending this information to server S2. This value does not change with increasing state sizesand the apparent increase seen in the column for this value is due to noise. The value seems to be between 85 to 90 microseconds for all state sizes in the range 1K to 14K bytes.
  - *Twait_migr*: The time taken at the client waiting for the migration to complete. The time is measured from the point the client sends the SYN with a special option to server S2 upto the point the SYN+ACK is received from server S2. We see a steady increase in this value with increasing state sizes. This is intuitive as increase in the state size leads to an increase in the time to migrate a connection.
  - *Tcomplete* : The time taken to complete the switch from the old server to the new server after the SYN+ACK has been received from server S2. This involves copying of options and other relevant information from the old connection endpoint's TCP control block to the new. This value does not depend on the state size and is

corroborated from the values shown in the table.

- **Server S1 side**
  - *Tprep_sr*: The time taken at server S2 to prepare a state-request for the connection to server S1. This value again does not depend on the state size and is between 40 to 42 microseconds.
  - *Twait_sr*: The time spent at server S2 waiting for the state-reply from server S1. This increases with the state size as the size of the state reply increases linearly with state size.
  - *Treinstate_s*: The time taken at server S2 to reinstate the connection endpoint from the state information sent by server S1. This value is between 20 and 27 microseconds and is independent of the state size.
- **Server S2 side**
  - *Tprep_s*: The time taken at server S1 to prepare the state-reply for a request. This value is dependent on the state size as this involves a copy of the state from the connection endpoint onto the state-reply message.

The graph in Figure 5.3 shows the dependency between the state size and the time to migrate. The graph validates the intuition of a linear dependency between the state size and the time to migrate.

5.3.2 Experiment II: *Stream Server Experiment*

The second experiment was performed with an aim to test the mechanism with a real and common server application whose performance degrades with time. The generic application chosen was a stream server, similar to a web server, where the server keeps pushing data to the client after an initial request by the client. The experimental setup was exactly as described in fig.5.1. The server sends data chunks of size 1024 bytes to the client. The total length of the stream in the experiment was 256K bytes. After writing every chunk of data the server application takes a snapshot. The application-level snapshot consists of the *chunk number* of the last chunk of data written by the server to the buffers. To *simulate* the scenario of a

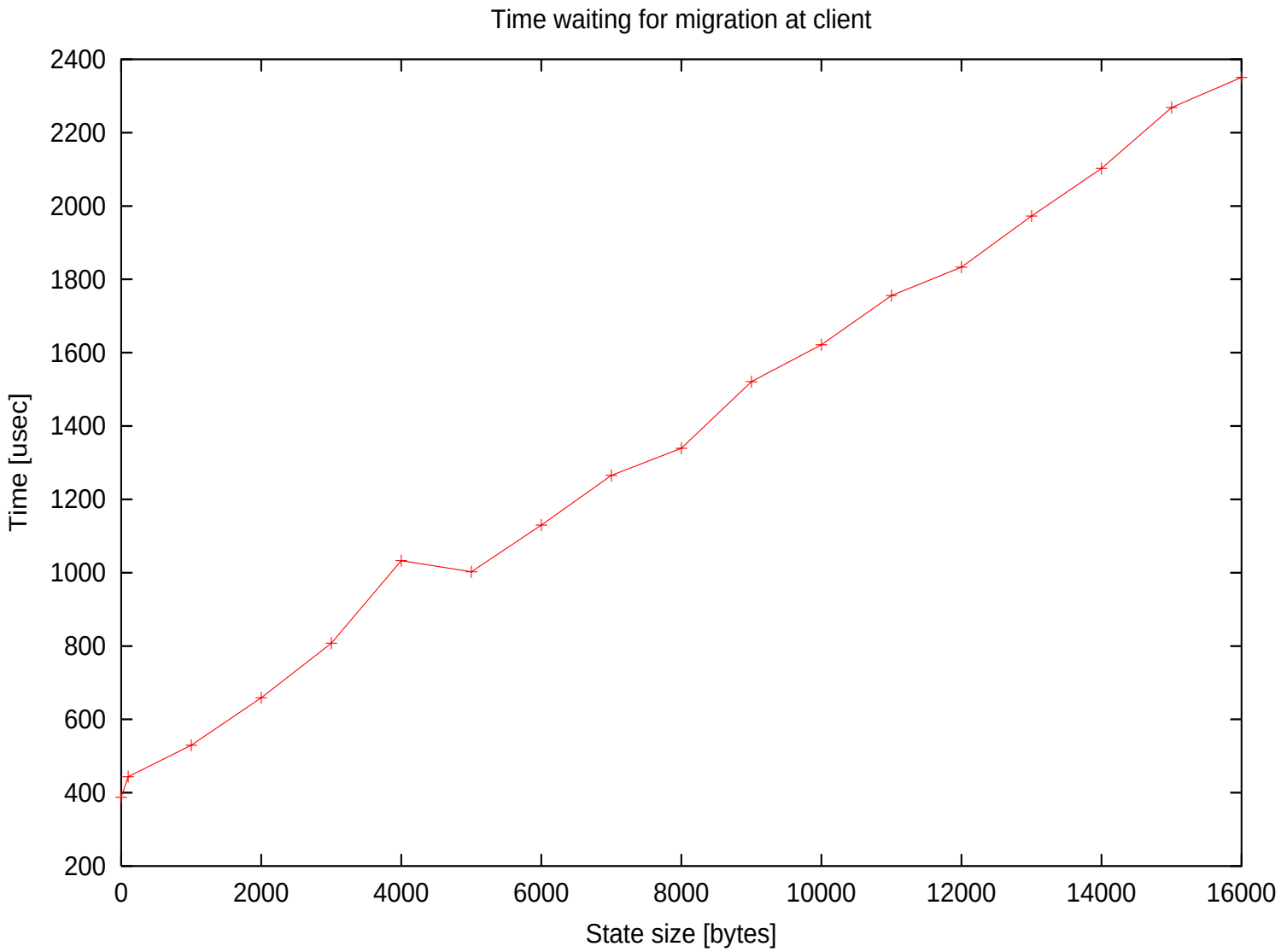

Figure 5.3: Time to migrate versus connection state size

degrading server, the server application artificially introduces delays in the writing of chunks after a threshold value. This affects the throughput as seen from the client and this triggers the migration.

The triggering itself is done by a policy module inserted into the kernel on the client side for this experiment. This policy can be refined and used for this class of applications (streaming servers). This module maintains a smoothed value of the effective throughput perceived at the client side. This value is computed as follows:

*S_Throughput = (α * C_Throughput) + (1 - α) *S_Throughput)*.

The current throughput (C_Throughput) is computed every time a new packet is received on the connection. Smoothed Throughput (S_Throughput) is computed using the current throughput by using the low-pass filter formula shown above with an α value of 0.4. This might not be a good means to find the smoothed value of throughput, but we chose this for its simplicity and low overhead in computation. As the delays engineered at the server end were gradual, this formula gave us a good measure of the effective throughput. In addition to this the maximum value of the smoothed throughput is maintained. The policy module decides to trigger a migration when the smoothed bandwidth reaches 0.7 times the maximum smoothed throughput. The second server now responds to the client at a faster rate initially (before the delay threshold is crossed) and then the same cycle repeats. This experiment was conducted for three different sizes of the application state:2KB, 10KB, 16KB. The results are shown in the graph (Figure 5.4).

The graph plots increasing sequence numbers received from the server by the client with time. The sequence numbers plotted are not the absolute values but are relative to the IRS (*Initial Receive Sequence*) number. The highest sequence number that can be received is 256K. The lowest curve represents the default behavior of the server. It initially gives a good throughput (as seen from the initial slope), then the rate gradually decreases. The other curves shown are for differing state sizes with migration enabled.

Consider the curve belonging to an application state size of 10K i.e. the one that is just above the lowest curve. Each *arc* seen on the curve corresponds to one interaction of the client with a particular server. The client migrates back and forth between S1 and S2 during the session resulting in the different *arcs*. In a way, the client stays with a server only during the

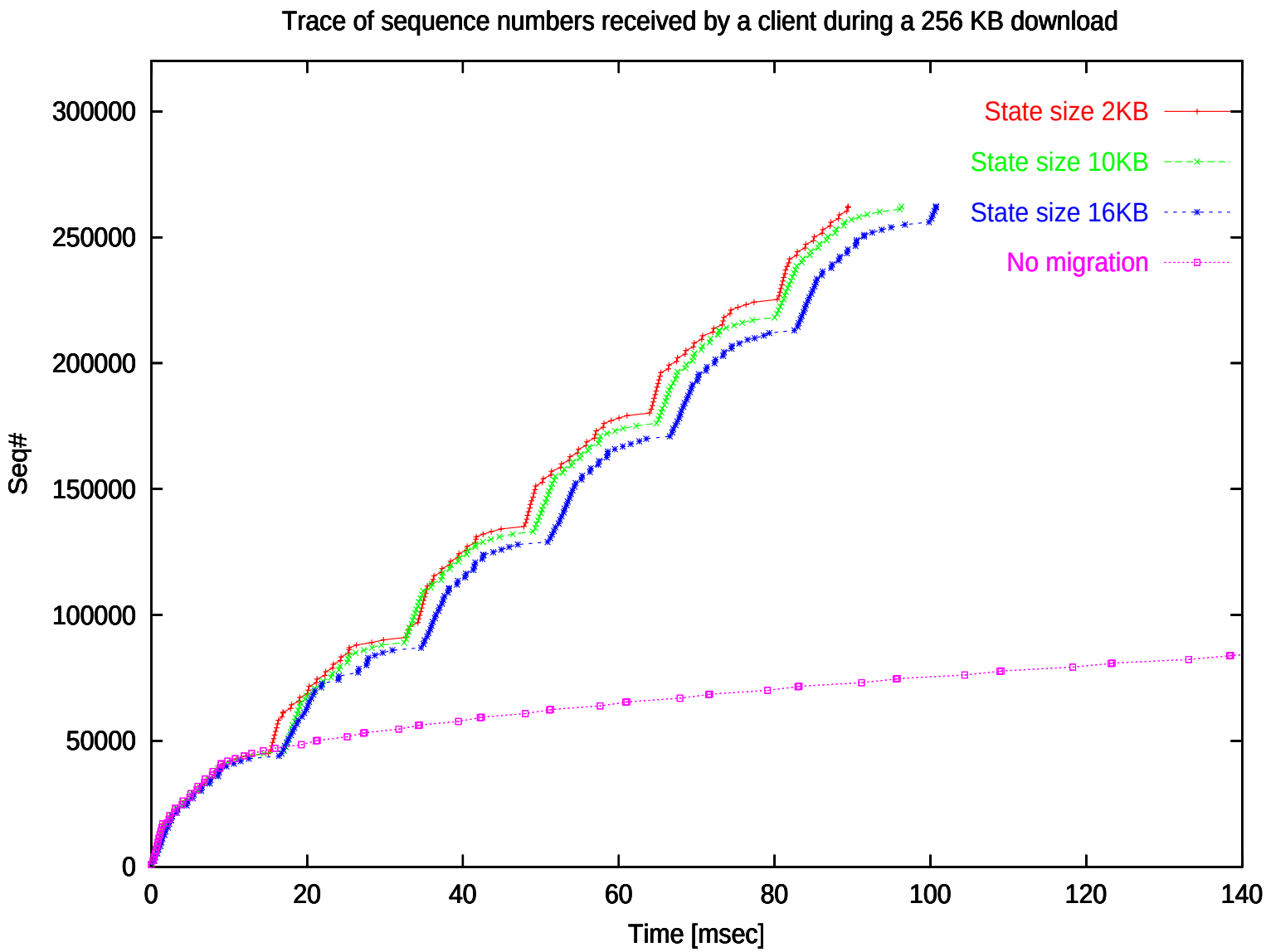

Figure 5.4: Stream server experiment

time when it receives data at a good rate from the server. In summary, from the top three curves we observe that the *effective rate at which the client receives the data is almost close to the best rate at which the server performs*.

The effective throughput with a lower application state size is higher compared to a larger application-state size because of two reasons:

- The state transfer time is directly dependent on the size of the application state size.
- The time to take a snapshot is directly related to the size of the application state size.

This is clearly illustrated from the first three curves. The client connected to a server application that has a size of 2KB finishes receiving the stream first, followed by the 10KB and the 16KB ones.

Chapter 6

Protocol Utilization and Applications

In this chapter we elaborate on the applications built over our migration mechanism. In the following section, we illustrate the modifications necessary to a server application by means of an example. This is followed by a section explaining the different classes of applications that could utilize our connection migration mechanism.

6.1 Migration-Enabled Server Applications

In this section we briefly describe the changes that a server application needs to incorporate to make it migration-enabled followed by a concrete example that illustrates these changes.

6.1.1 Modifications

Our approach involves some changes to be incorporated in the design of the server application. The server application should obey a certain programming discipline by calling primitives for exporting/importing to/from the OS the application level state associated with a potentially migrating connection. However, the client application does not require any changes.

The application may have to exploit certain tradeoffs. One possible tradeoff is between the size of the *state snapshot* (considering both in-kernel and application level state) and the moment the *snapshot* is taken. Another tradeoff to be investigated while designing the server application is the size of the application state itself; in some cases increasing the size may capture more computation and might result in *less computation to be redone after resuming* at the new server after a migration. However, larger state sizes would lead to a larger time for migration as the overhead due to network transmission time is directly proportional to the state size to be transferred.

Although we require changes to the server applications, we believe that in general the programming effort involved should be fairly low. 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 endpoint migrations.

6.1.2 An Example Server Application

A simple web server application is chosen to illustrate the modifications required to make a server application migration-enabled (Figure 6.1). The per-connection application state information for this application consists of the file name (`state.fname`) requested and offset in the file (`state.off`). The ordinary (without migration support) server application is shown in the left half of fig.6.1

and the migration-enabled server is shown in the right half.

In the ordinary application, after a successful `accept` call the following actions take place: (the line numbers correspond to those in fig. 6.1)

1. The request sent by the client is parsed and the file name requested is extracted (lines 1-4).
2. The file is opened (line 5).
3. Data from the file is read and written to the socket till the end of the file is reached (lines 6-13).

In the case of the migration-enabled server application, after a successful `accept` the following actions take place:(the line numbers correspond to those in fig. 6.1)

1. A `get_state` call is made to determine the nature of the connection. In case of a new connection, the `get_state` call returns `NEW`. In case of a migrated connection from a different cooperating server, the `get_state` call returns MIGRATED and the associated application level state is available in the buffer `state` sent as a parameter in the `get_ state` call (line 2).
2. In case of a new connection, the state information is initialized after parsing the file name from the request. The `state.off` value is set to zero to signify that no data has been sent

```
1  s = accept(ssock)
     .
     .
2  // ... parse request, get fname
3  state.fname =  fname
4  state.off = 0
5  fd = open(fname)
     .
     .
6    // in a loop, send from file...
7    while( !endoffile(fd)) {
       .
       .
8      // file read
9      n = read (fd, &buf, nmax)
10     // write to socket
11     write (s,  &buf, n)
12     state.off += n
       .
       .
13   }
```

A. Without Migration Support

```
1  s = accept(ssock)
2  kind = get_state(s, &state)
3  if (kind == NEW) {
4    //  new connection: start
5    // ... parse request, get fname
6    state.fname =  fname
7    state.off = 0
8    set_state(s, &state)  // snap
9    fd = open(fname)
10 } else if (kind == MIGRATED)
11 {
12   // migrated connection: resume
13   fd = open(state.fname)
14   lseek(fd, state.off, SEEK_SET)
15 }
16 // both new and migrated
17 // in a loop, send from file...
18 while( !endoffile(fd)) {
       .
19   // file read
20   n = read (fd, &buf, nmax)
21   // write to socket
22   err = write (s, &buf, n)
23   if (err = EMIGRATED) {
24     exit ()
25   }
26   state.off  += n
27   set_state (s, &state) // snap
       .
25 }
```

B. With Migration Support

Figure 6.1: Examples of server applications

on the connection. The first state snapshot is taken by means of a `set_state` call. The file is then opened (lines 3-9).

3. In case of a migrated connection from a cooperating server, the file name and file offset are extracted from the `state` value. The appropriate file (`state.fname`) is opened and the file pointer is moved to the proper position (as denoted by `state.off`) (lines 10-15).

4. The remaining set of actions are common to both new and migrated connections. Data is read from the file in `buf` and written to the socket (lines 19-22). The `state.off` value is modified to reflect the number of bytes read from the file (`n`). If the write to the socket was successful then a state snapshot is taken with the new state by means of a `set_state` call (line 27). A snapshot is taken at this point because the application level state is consistent with the data written to the buffers. The write to a socket might not be successful if the connection has migrated to another server, in this case the application simply exits (lines 22-25). This process repeats till the end of the file is reached.

In the example chosen above, the per-connection application state is well-defined and small. This makes the task of making the server application migration-enabled easier. However, this example clearly illustrates the programming model and discipline to be followed while designing server applications.

6.2 Applications

In this section we describe the applications that could benefit from our connection migration mechanism. The primary among these are the use of our mechanism in facilitating fault tolerance, high availability and load balancing schemes.

6.2.1 Applications in Fault Tolerance

Using Figure 3.2 as reference, we can think of a scenario where S1 dies abruptly taking the live service sessions along with it. If S1 was able to transfer the state pertaining to the some/all of the live connections to another cooperating server, then we can salvage those connections using our mechanism. This requires the state to be present at S2 when the request for the migration

comes from the client. An interesting alternative is to store the server state at the client. The client can then cooperate with the new server and reincarnate the server endpoint.

6.2.2 Applications for High Availability

For many Internet services, availability of the service would be a core concern for the clients. To make the service more available, the service should be able to withstand *DoS attacks* (Denial of Service) and overload. Efficient means to detect a DoS attack coupled with our mechanism would make the service highly available. As soon as a *DoS attack* is detected, several of the existing connections can be migrated to a cooperating server to ensure continued service. To protect against overload, a good load-balancing scheme needs to be implemented in association with our mechanism.

The main advantage of our scheme with respect to other schemes is our ability to move service sessions dynamically and at any point of time during the course of the connection. Consider a scenario in which the load at a server due to a particular session changes dynamically with time. Then it is possible that existing connections might cause the overload to the server even though there are no new connections to the server. This situation cannot be solved by static load-balancing schemes that distribute the load on the servers by allocating new connections to least loaded servers.

6.2.3 Applications in Load Balancing schemes

Suppose we have a scheme in which servers act as triggers for migration (Figure 3.2). Each connection is an atomic unit in the load experienced by the server. We can think of a load-balancing scheme being implemented among the servers by dynamically migrating connections between them. Once a load-balancing decision is made at a server, it just becomes a migration trigger for a connection, and sends a *MIGRATE_TRIGGER* segment to the client. Then the client initiates the migration by connecting to the destination server. Two styles of load balancing interactions are possible, depending on where the load-balancing decision is made.

1. S1 could decide to push some connection away to S2.

2. S2 could decide to pull some connection from S1. In this case S2 must know, through

some external mechanism, the *Cid* of the connection it wants to pull from S1.

In both cases the trigger's decision is independent of and transparent with respect to respect to the client-initiated migration mechanism. Moreover, the scheme can be extended to allow another host, different from the origin and destination servers to act as a migration trigger.

Chapter 7

Conclusions and Future Directions

We have described the design and performance of a transport layer infrastructure - MTCP (Migratory TCP). MTCP provides a framework for building highly available services over the Internet. The need for such a transport layer support is apparent from the emerging classes of applications being built over the Internet. We also exploit the recent paradigm shift in viewing resources over the Internet more as services than as servers. The infrastructure facilitates high availability by enabling dynamic connection migrations between cooperating servers providing the same service. Our solution is generic and can help in building fault-tolerant systems and can also facilitate load-balancing among servers.

The salient features of our system are:

- Our mechanism allows dynamic connection migration at any point of time during a service session. Migration of the connection between peer servers can happen any number of times during the session. The *cooperating peer servers* might be geographically distributed across the Internet.

- Our connection migration mechanism is totally transparent to the client applications. However, this requires a change in the transport layer protocol being used at the client end.

- Our mechanism does not take into account any higher level protocol/application specific information to bring about connection migration. Server side applications need to be rewritten to follow a programming model and also have to make use of our API to incorporate high availability into their services. In addition, the server side transport layer protocol also needs modification.

- A wide range of policies can be employed in our connection migration model. We have

designed and built a mechanism for migrating connections leaving policy decisions out of the model.

- The key features of TCP have not been modified to achieve dynamic connection migration. Any further enhancements to TCP would not affect our mechanism but would accrue all the advantages provided by them. As far as our knowledge, the basic semantics of TCP have not been violated during the connection migration process.

We have implemented MTCP by modifying the TCP/IP stack of the FreeBSD Operating System. The experimental results shown validate the feasibility of our connection migration model. As a part of our experiments, we have built a simple *stream server* application, similar to a web server employing our migration mechanism. This application serves as a model to build other applications over our transport layer protocol. We need to devise more experiments to test our mechanism in environments closer to those existing in the current Internet.

The current implementation employs a *lazy* connection state transfer scheme. This has certain limitations as it cannot ensure state transfer under all circumstances. Thus the applications of this implementation are limited to certain domains only.

An *eager* connection state transfer implementation of our mechanism would be highly applicable in many more areas. Building an *eager* connection state transfer implementation would be one of the first tasks to follow on the current work. Porting this work onto the Linux kernel would be a worthwhile exercise given the number of applications and popularity of the operating system. Some new transport layer protocols oriented towards providing high availability and fault tolerance like SCTP (Stream Control Transmission Protocol) have been proposed. One of the future tasks would be to port our migration mechanism over to SCTP, and analyse the new possibilities that arise from this synergy.

Efforts are on to build a migration-aware full-fledged transaction server application with a database system in the back-end. The goal of this system would be to provide transactions on a database that would be tolerant to loss of network connectivity to/failure of a front-end to the DB system. A basic version of this system has been implemented and extra features are currently being added to the system.

In the current design, a basic assumption is that the per-connection state is represented as a

chunk of memory in the application. This may not be entirely true in some applications where one connection has close dependencies with a set of other connections. In these cases, the other connections are logically part of the per-connection state. Thus transferring the per-connection state between cooperating servers would also involve *recursively* migrating all the dependent connections.

References

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

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

[3] F. Sultan, K. Srinivasan, L. Iftode. Transport Layer Support for Highly-Available Network Services. HotOS-VIII, May 2001.

[4] F. Sultan, K. Srinivasan, L. Iftode. Transport Layer Support for Highly-Available Network Services. DCS TR-429, Rutgers University.

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

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

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

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

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

[10] C. Yang, M. Luo. Realizing Fault Resilience in Web-Server Cluster. SuperComputing 2000, November 2000.

[11] V. Jacobson. Congestion Avoidance and Control. In Proceedings of ACM SIGCOMM, Pages 314-329, August 1988.

[12] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, May 1992.

[13] 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, Oct.1995.

[14] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. ACM Computer Communications Review, July 1996.

[15] J. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In Proceedings of the ACM SIGCOMM'96, Aug. 1996.

[16] D. Lin and H.Kung. TCP Fast Recovery Strategies: Analysis and Improvements. Proceedings of the IEEE INFOCOMM '98, San Francisco, Mar. 1998.

[17] M. Mathis and J. Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. Proceedings of the ACM SIGCOMM '96, Aug. 1996.

[18] Z. Wang and J. Crowcroft. A New Congestion Control Scheme: Slow Start and Search (Tri-S). ACM Computer Communications Review, Jan. 1991.

[19] 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.

[20] C. A. Thekkath, T. D. Nguyen, E. Moy and E. D. Lazowska. Implementing network protocols at user level. In Proceedings of ACM SIGCOMM, pages 64-73, September 1993.

[21] D. Clark and D. Tennenhouse. Architectural Consideration for a New Generation of Protocols. In Proceedings of ACM SIGCOMM, September 1990.

[22] W. R. Stevens and G. W. Wright. TCP/IP Illustrated. Vols. I, II and III. Addison Wesley Publications.

[23] D. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In Proceedings of IEEE Infocom '98, March 1998.

[24] Mohit Aron, Darren Sanders, Peter Druschel, Willy Zwaenepoel Scalable Content-aware Request Distribution in Cluster-based Network Servers USENIX 2000, June 2000

[25] D. Maltz and P. Bhagwat. TCP Splicing for Application Layer Proxy Performance Technical Report RC-21139, IBM, March 1998.

[26] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. In Proceedings of USITS '99, October 1999.

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

[28] Information Sciences Institute, University of Southern California. Transmission Control Protocol. RFC 793, DARPA, September 1981.

[29] R. R. Stewart, Q. Xie, M. Tuexen, I. Rytina SCTP Dynamic Addition of IP addresses. Internet Draft, IETF, Nov.2000.

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

[31] G. Hunt, E. Nahum, and J. Tracey. Enabling content based load distribution for scalable services. Technical Report, IBM T.J. Watson Research Center, May 1997.

[32] E. Anderson, D. Patterson, and E. Brewer, The MagicRouter: an Application of Fast Packet Interposing". In Proceedings of OSDI, October 1996.

[33] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, Y. Wang. ONE-IP: Techniques for Hosting a Service on a Cluster of Machines Hyperproceedings of the Sixth World Wide Web Conference, April 2001.

[34] Foundry Networks Inc. ServerIron Family of Internet traffic and content management. http://www.foundrynetworks.com

[35] Cisco Systems Inc. Catalyst 6500 backbone switch. http://www.cisco.com

[36] E. N. Elnozahy, L. Alvisi, D.B. Johnson, Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *Technical Report CMU-CS-99-148*, Carnegie Mellon University, June 1999.

[37] J. F. Bartlett A NonStop Kernel Proc 8th SOSP, 1981.

[38] B. Walker, G. Popek, R. English, C. Kline, G. Thiel The LOCUS Distributed Operating System. Proceedings of the 9th SOSP October 1983

[39] A. Barak, S. Guday, R. Wheeler The MOSIX Distributed Operating System Spriger Verlag, 1993

[40] M. Litzkow, M. Solomon Supporting Checkpointing and Process Migration outside the UNIX Kernel. Proceedings of the USENIX Winter Conference. January 1992

[41] F. Douglis and J. Ousterhout Process Migration in the Sprite Operating System Proc of the 7th Conf on Distributed Computing Systems September 1987

[42] E. Pinheiro and R. Bianchini. Nomad: A Scalable Operating System for Clusters of Uni and Multiprocessors. Proceedings of the 1st IEEE International Workshop on Cluster Computing, December 1999.

[43] D. Milojicic, F. Douglis, Y. Panedeine, R. Wheeler, and S. Zhou. Process Migration. ACM Computing Surveys 32(3), pp. 241-299, September 2000.