CLIENT MIGRATION IN A CONTINUOUS DATA NETWORK

by

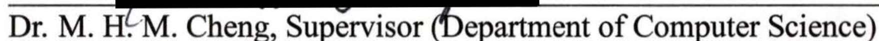
Anthony Justin Howe
B.Sc., University of Victoria, 1999

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

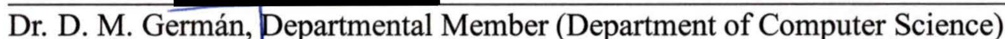
MASTER OF SCIENCE

in the Department of Computer Science

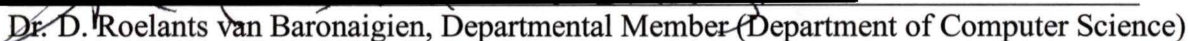
We accept this thesis as conforming to the required standard



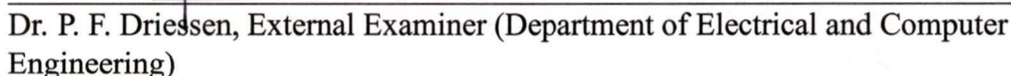
Dr. M. H. M. Cheng, Supervisor (Department of Computer Science)



Dr. D. M. Germán, Departmental Member (Department of Computer Science)



Dr. D. Roelants van Baronaigien, Departmental Member (Department of Computer Science)



Dr. P. F. Driessen, External Examiner (Department of Electrical and Computer Engineering)

© ANTHONY JUSTIN HOWE, 2002

University of Victoria

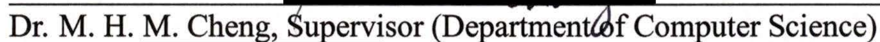
All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.

Supervisor: Dr. M. H. M. Cheng

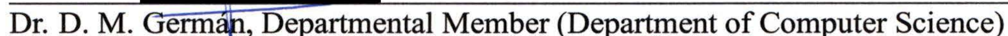
ABSTRACT

Two major issues for continuous data delivery architectures on the Internet are scalability and data continuity. Scalability is addressed by providing multiple redundant stream sources. One solution to the data continuity problem is adaptive client migration. Adaptive client migration is the ability of a client to switch from one redundant stream source to another when it experiences discontinuities in its current stream. The switching mechanism allows the service providers to adapt to changing server loads and varying jitter due to network congestion. This thesis presents an adaptive client migration protocol that allows a client to migrate from one data stream to another without introducing discontinuities.

Examiners:

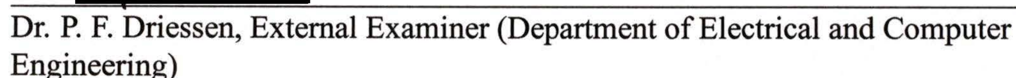


Dr. M. H. M. Cheng, Supervisor (Department of Computer Science)



Dr. D. M. German, Departmental Member (Department of Computer Science)

Dr. D. Roelants van Baronaigien, Departmental Member (Department of Computer Science)



Dr. P. F. Driessen, External Examiner (Department of Electrical and Computer Engineering)

Table of Contents

Table of Contents iii

List of Figures v

List of Tables vii

Acknowledgments viii

Dedication ix

1 Introduction 1
- 1.1 Continuous Data Delivery 2
- 1.2 Common Network Architectures 2
- 1.3 Contributions of this Research 5
- 1.4 Related Work 7
- 1.5 Thesis Overview 7

2 A Continuous Content Distribution Network Model 8
- 2.1 A Model of a Generic Content Delivery Architecture 8
- 2.2 Delivery of Discrete Data 10
- 2.3 Delivery of Continuous Data 12

3 The Client Migration Protocol 14
- 3.1 A Sliding Window 14
- 3.2 Client Migration 17
- 3.3 Protocol Design 21
- 3.4 Fault Analysis 24
  - 3.4.1 Control Message Faults 26
    - 3.4.1.1 Lost Control Messages 26
    - 3.4.1.2 Duplicate Surrogate Migration Requests 28
    - 3.4.1.3 Logout from a Surrogate During Migration 28
  - 3.4.2 Data Message Faults 29
    - 3.4.2.1 Buffer Underruns and Overruns 30

4 Experimental Validation 32
4.1 Experimental Framework . . . 32
4.2 Distributed Initiation . . . 34
4.3 Idempotency . . . 35
4.4 Resiliency . . . 36
4.4.1 Varying Surrogate Latencies . . . 37
4.4.2 Invalid Surrogate Latencies . . . 38
4.4.3 Control Message Faults . . . 39
4.4.3.1 Lost Control Messages . . . 40
4.4.3.2 Duplicate Surrogate Migration Requests . . . 42
4.4.3.3 Logout from a Surrogate During Migration . . . 42
4.4.4 Data Stream Faults . . . 43
4.5 Performance . . . 44
4.5.1 Performance Results . . . 44
4.5.2 Light Weight . . . 47

5 Concluding remarks 49
5.1 Future Work . . . 49

References 51

Appendices 52

A The Migration Protocol 53
A.1 Coordinator . . . 53
A.2 Surrogate . . . 57
A.3 Client . . . 60

List of Tables

4.1 Summary of migration timing results 46

A.1 Summary of coordinator messages and transitions 55

A.2 Summary of the timers used by the coordinator 56

A.3 Summary of the variables used by the coordinator 56

A.4 Summary of surrogate messages and transitions 58

A.5 Summary of the timers used by the surrogate 59

A.6 Summary of the variables used by the surrogate 59

A.7 Summary of client messages and transitions 61

A.8 Summary of the timers used by the client 64

A.9 Summary of the variables used by the client 64

A.10 Summary of the client buffer functions 65

List of Figures

1.1 An example of a server farm 3

1.2 An example of a proxy cache 4

1.3 An example of a content distribution network 5

2.1 A model of a generic content delivery architecture 9

2.2 The transfer of discrete data 11

2.3 The transfer of discrete data from multiple surrogates 11

2.4 Data channel and control channel connections 13

2.5 The transfer of continuous data from the origin to client 13

3.1 A sliding window B with a read and write head 15

3.2 The sliding window at time $t = 100$ and at time $t = 104$ 16

3.3 The actions of a read operation and a write operation on a sliding window . . 17

3.4 Data channel communication in a continuous data network 18

3.5 Surrogates and a client at various positions in a data stream 19

3.6 Client C's migration to the valid surrogates S_2 and S_3 of Figure 3.5 20

3.7 The message sequence chart for a client migration 22

3.8 The window showing the splicing of the two streams 23

3.9 Client migration to a surrogate too far upstream from the current stream . . . 24

3.10 The removal of a gap after a failed migration to a *later* stream 25

3.11 Non-monotonically increasing sequence numbers of the data stream 26

3.12 The message sequence chart for a client migration with timers 27

4.1 The Generic Data Delivery Application as it appears in COOL 33

4.2 The transfer of continuous data from the origin to the client 34

4.3 The message sequence chart showing multiple migration requests from a client and surrogate during a single client migration 36

4.4 The message sequence chart showing the migration to a surrogate of two data units downstream from the existing surrogate 38

4.5 The message sequence chart showing the migration to a surrogate two data units upstream from the existing surrogate 39

4.6 The message sequence chart showing client migration to a surrogate too far upstream (for simplicity control message communication to the surrogates has been removed) 40

4.7 The message sequence chart showing a lost ***register()*** message from C to S_2 . 41

A.1 The state-chart for the coordinator 54

A.2 The state-chart for the surrogate 57

A.3 The state-chart for the client 60

Acknowledgments

Financial Support for this research came from the Natural Sciences and Engineering Research Council (NSERC) of Canada, the Mathematics of Information Technology and Complex Systems (MITACS) of Canada, and Nortel Networks.

I would like to thank my supervisor, Dr. M.H.M. Cheng, of the Department of Computer Science, University of Victoria, for his constructive criticism and encouragement. I would also like to thank him for convincing me to get my Master's degree, in March 1999, when I had the difficult choice on whether to enter the work force or continue with my studies. I would like to thank all the members of my committee and my external examiner for their input into my thesis.

Many graduate students, including Gordon O'Connell, and Steven Shelford, provided valuable feedback during the development of my thesis.

I would like to thank Josephine for her constant support, patience, and encouragement throughout the last two and half years.

Finally, I would like to thank my Dad, Mom, and brother Kevin for their continuing encouragement throughout my school years. Without their guidance with public speaking and science fairs I may never have reached this level of achievement.

Dedicated to my Dad and Mom.

Chapter 1

Introduction

The increase of bandwidth on the Internet has led to a reduction of end-to-end delay in content delivery. More and more Internet applications are taking advantage of this low end-to-end delay and are delivering live content to tens of millions of users [1]. Examples of live content include Internet radio, voice chat, and live video broadcast [2] [3] [4]. Live content refers to continuous data. Continuous data requires the time sensitive delivery of data from a server to a client; late data is the same as lost data.

Several network architectures have been proposed for the purpose of delivering continuous data to end users on the Internet [5] [6] [7] [8]. These architectures include server farms, proxies, and content distribution networks. A strength shared by each of these architectures is their ability to scale to support thousands or even millions of connections [9]. Scalability is achieved by replication of the source content to multiple continuous data servers. A common problem shared by each of these architectures is their lack of guarantee for data continuity. Network congestion or overloaded servers may cause undesirable breaks or dropouts in continuous data streams.

This thesis addresses the data continuity problem of continuous content delivery by presenting adaptive client migration. Adaptive client migration allows a client to migrate to another continuous data stream after detecting jitter or discontinuities in its current data stream. Using adaptive client migration a continuous data delivery architecture is able to adapt to changing server loads and varying jitter due to network congestion.

1.1 Continuous Data Delivery

The delivery of continuous data on a network is reliant upon the following attributes: bandwidth, latency, jitter, compression and decompression algorithms, and network architecture [10] [5] [11]. The first three attributes are quality of service (QoS) properties; they are handled in the first three layers of the OSI model [12] [13]. The final two attributes are dealt with in layers 4 through 7 of the OSI model [12] [13].

QoS properties are concerned with how much, how fast, and how timely data is delivered from one location to another. Bandwidth refers to the amount of data transferred in a given amount of time. Without sufficient bandwidth a client receiving continuous data may not have enough information to construct a satisfactory reproduction of the original source. Latency refers to the time a piece of data takes to get from the source to the destination. Low latency is critical for a client to provide responsive feedback to the server transmitting the continuous data. Jitter is the variation of time between the delivery of a series of data messages. Minor jitter problems can be ironed out with the use of buffers. Major jitter problems will cause discontinuities in the continuous data.

Lossy or lossless data compression and decompression algorithms are used to reduce the size of the continuous data [14]. The compression and decompression algorithms must have high enough performance so that the rate of compression and decompression of the continuous data exceeds or matches the delivery rate of the original source.

1.2 Common Network Architectures

A single server can be used for the delivery of continuous data to clients. However, the audience size of the single server is limited by the server's computing resources and the available network bandwidth. To increase the available connections to a streaming source more servers may be added. Three common network architectures constructed from the addition of more servers to a single server are server farms, proxies, and content distribution networks.

Server farms rely on an intelligent switch to evenly distribute requests among a group

of servers hosting the continuous data [9] [6]. A server farm appears to the user as a single server. Figure 1.1 shows a server farm. An intelligent switch evenly distributes clients among the servers. The dotted square outline represents the fact that the servers and switch are in close location to each other. A server farm allows for a larger audience to receive the same continuous data than could be provided by a single origin. The main disadvantage of a server farm architecture is that it has no control over the network between the server farm and the client. A client many network hops away from the server farm may not receive jitter-free delivery of a data stream due to unpredictable congestion on one of the network hops.

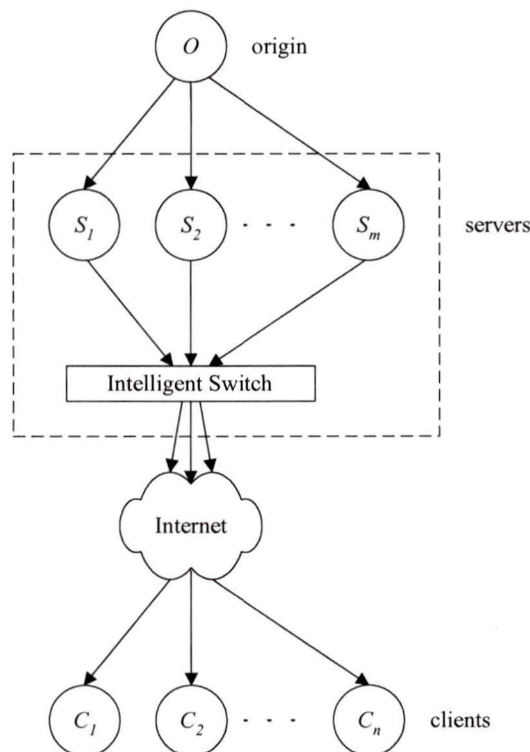


Figure 1.1: An example of a server farm

Proxies, also known as proxy caches, attempt to bring continuous data closer to the end user [9] [7]. Figure 1.2 shows a proxy cache network. The clients on the same or nearby networks pass all their continuous data requests through a proxy. A dotted square

outline shows that a proxy and its client are in close location to each other. If the proxy is already hosting the requested data stream it will deliver the same stream to the requesting client. Otherwise, the proxy will spawn a new data stream from the origin server and then deliver that stream to the client. A proxy ensures that only one data stream is set up between itself and an origin server. This reduces load on an origin server. Since a proxy network architecture ensures clients are in close location to the proxy server, the clients will have a higher probability of receiving jitter-free continuous data from the proxy. However, like the clients of a server farm, the proxy can be many network hops away from the origin of a data stream and may not receive jitter-free delivery of the data stream.

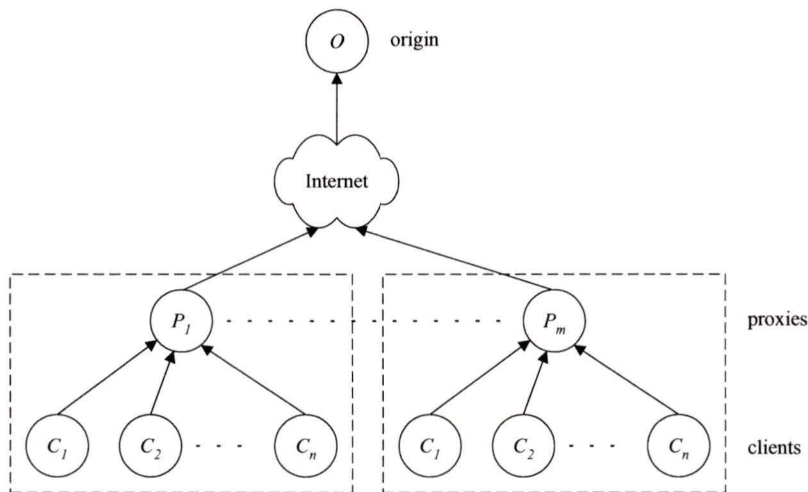


Figure 1.2: An example of a proxy cache

A content distribution network combines the advantages of server farms and proxy caches [9] [8]. Figure 1.3 shows a content distribution network. Continuous data streams are set up on dedicated network resources between an origin and servers known as surrogates. The dedicated network resources are shown with the heavy line and represent the fact the the timely continuous data delivery is guaranteed. The surrogates are located geographically far apart and may form a server farm themselves. When a client requests a continuous data connection it first communicates to the coordinator. The coordinator then determines the *best* surrogate for the client. The measure of *best* can be derived from ge-

ographic location, current network load and congestion, and available surrogate resources. A connection is then set up between the client and its *best* surrogate. The surrogate then delivers the continuous data to the client. Even though the client is connected to the *best* surrogate, the client may experience jitter caused by network congestion or surrogate overload. The *best* surrogate may change over time.

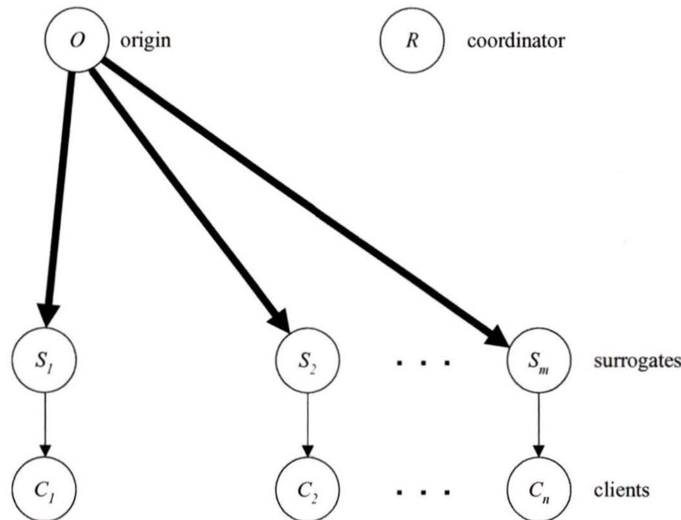


Figure 1.3: An example of a content distribution network

1.3 Contributions of this Research

The previous section described three network architectures and their ability to scale to support many more client connections than could be supported by a single server. However, it was shown that each architecture suffered from the inability to guarantee uninterrupted continuous data. Continuous data could be slowed or interrupted by network congestion or overloaded servers.

The goal of this research is to develop adaptive client migration to overcome this data continuity problem. Adaptive client migration is the ability of a client, server, or coordinator to recognize congestion or server overload and switch clients to servers that reside

in a non-congested area of the network or are less loaded. This ability of clients to switch between continuous data servers assumes that there are two or more redundant streams. The three network architectures discussed in the previous section fulfill this assumption.

Adaptive client migration is achieved through the development of a client migration protocol. Our approach has the following three properties: distributed initiation, idempotency, and resiliency.

distributed initiation Distributed initiation is the ability of any entity, except the origin, in the network to initiate migration. A coordinator may initiate a client to migrate after monitoring network load. A server may initiate migration if it is overloaded. A client could detect a deterioration of the quality of its data stream and initiate migration.

idempotency Due to the nature of the client migration protocol, multiple requests for migration may occur simultaneously. Our protocol is idempotent and can take multiple requests for migration that result in a single migration. This means that a migration in progress is not affected by other migration requests. After migration a request to migrate to the same server will be ignored.

resiliency The migration protocol is resilient and can handle varying surrogate latencies, lost control messages, and some data stream faults. The migration protocol can splice a new data stream that is upstream or downstream from the current data stream. A data stream is upstream (in the future) from another data stream if it experiences less latency to the origin. Likewise a data stream is downstream (in the past) from another data stream if it experiences more latency to the origin. If the new stream is too far upstream or downstream to fit within the bounds of the client's buffer the migration protocol will abort the migration and report to the coordinator. Furthermore, lost control messages will not cause any network entity to deadlock or lose resources.

1.4 Related Work

The *Split and Merge Protocol* of Wanjiun Liao and Victor O. K. Li [15] is different from the client migration protocol presented in this paper. The *Split and Merge Protocol* used multiple readers and one writer to reduce the amount of bandwidth used in a continuous data network. The client migration protocol is the opposite and had multiple writers, the surrogates, and a single reader, the client. The *Split and Merge Protocol* assumed a high speed, high bandwidth local area network and did not deal with data discontinuities or jitter.

The paper *Distributed Video Streaming Over Internet* by Thinh PQ Nguyen and Avideh Zakhor describes a protocol to deliver a video stream from multiple senders to a single receiver [16]. The purpose of the protocol is to reduce packet loss and jitter by interleaving multiple streams. No two senders send the same media packet. The receiver coordinates each sender using a packet partitioning algorithm. This approach differs from our approach since we assume each sender sends the same continuous data stream.

A new architecture for delivering streaming media named Allcast has recently appeared [17]. Allcast starts with an origin streaming server and then grows as more clients connect to the network. Each client may be connected to the origin or another client receiving the stream. When a client leaves the network, Allcast must move any connected clients to other clients or the central origin. The problems Allcast addresses are very similar to ours. We cannot compare our approaches due to lack of published technical details.

1.5 Thesis Overview

Chapter 2 introduces a model of a generic data delivery architecture. The model describes the procedure necessary for a client to receive data from a server. It then discusses the challenges of the model to handle the delivery of continuous data. Chapter 3 presents the design of our client migration protocol. Chapter 4 describes our implementation of the client migration protocol, provides experimental validation for the three properties discussed above, and presents a performance evaluation of the protocol. Finally, Chapter 5 summarizes the results and contributions of this thesis and then suggests areas for future work.

Chapter 2

A Continuous Content Distribution Network Model

This chapter presents a model of a generic content delivery architecture for both discrete and continuous data. A general overview of the generic content delivery architecture is introduced, followed by a description of discrete data delivery, and concluded with a description of continuous data delivery. The requirements for both discrete and continuous data delivery are presented. The continuous data delivery aspect of this architecture is used in later chapters for the development and integration of the client migration protocol.

2.1 A Model of a Generic Content Delivery Architecture

The generic content delivery architecture, shown in Figure 2.1, is composed of four elements: the coordinator, the origin, the surrogate, and the client. The coordinator provides login and logout, data registration, and data discovery mechanisms for each of the other elements. An origin is the original source for data. A surrogate is an intermediate entity that contains a replica of data from the origin. The purpose of the surrogates is for scalability. The more surrogates in the architecture, the more clients it will be able to handle simultaneously. A client uses the discovery mechanism of the coordinator to discover the location of a surrogate that contains desired content. The surrogate delivers the content to

the client. Similar terminology of coordinator, origin, surrogate, and client is defined in [9].

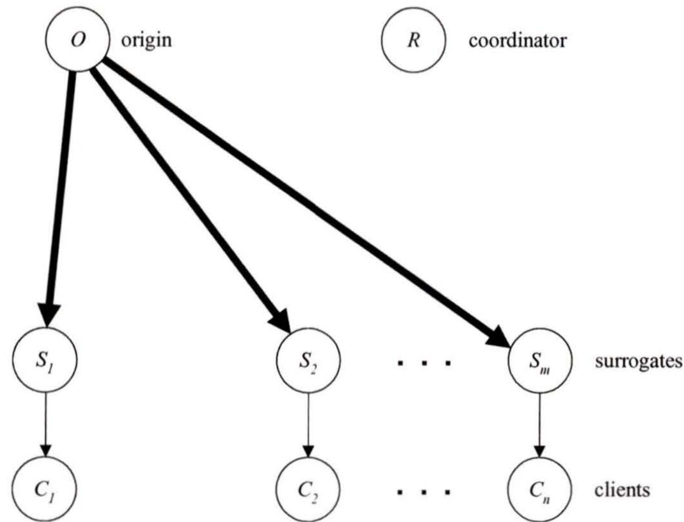


Figure 2.1: A model of a generic content delivery architecture

The elements of the generic content delivery architecture are connected via asynchronous channels; messages are ordered but unreliable. All elements in the architecture communicate with each other through *bi-directional control* channels. In addition the origins, the surrogates, and the clients transfer or receive data through *uni-directional data* channels. Each element in the network has a unique identification number (ID). The coordinator has a well known ID that every element is able to use for communication.

The origins, surrogates, and clients must authenticate with the coordinator before using the mechanisms of the coordinator. Once authenticated, the origins and surrogates register all the data they contain to the coordinator. The coordinator uses the data registration information to aid in the discovery and delivery of data for a client.

For a client to obtain data it must go through a discovery phase to determine where data is located and then connect to a surrogate that has the data. The following summarizes the data discovery and delivery steps.

1. Client C requests data d from coordinator R.

2. R finds surrogates $\{S_1, S_2, S_3, \ldots, S_n\}$ that contain d.

3. R or C selects S_j such that S_j is the *best* surrogate to supply d to C.

4. C connects to S_j.

Steps 1-3 handle data discovery and step 4 handles data delivery. The measure of *best* when determining a surrogate for the client may be based on one or more of the following properties: the geographic location of a surrogate with respect to a client, the network locale of a surrogate and a client, the number of hops from the client to a surrogate, the current network resources of a surrogate, the current computing resources of a surrogate, or the quality of the *data* channel between a surrogate and the client. In choosing the *best* surrogate for a client our primary objective is to optimize the client's QoS.

2.2 Delivery of Discrete Data

By discrete data delivery, we refer to the transfer of non-time sensitive data from a surrogate to a client, i.e., each segment of data does not have to arrive at a fixed periodic rate. Examples of discrete data include text, image, and binary files. The requirement of discrete data is that the whole discrete data item will eventually arrive at a client; it is not important *when* the data item is received, provided that all of the data item will be received. Hence, guarantee of delivery is an important requirement.

Figure 2.2 shows the transfer of discrete data messages d_1 to d_n from surrogate S_1 to client C. As can be seen by the figure the data arrives to the client at an unpredictable rate, however all the data eventually arrives at the client.

The transfer of discrete data may come from multiple sources. For example in Figure 2.3 the client C receives d_1 to d_j from surrogate S_1 and then receives d_{j+1} to d_n from surrogate S_2. The amount of time between receiving data message d_j and d_{j+1} may be considerable due to the coordination of data delivery set up with surrogate S_2; this is acceptable as long as data is not time sensitive. To speed up a discrete data transfer the client

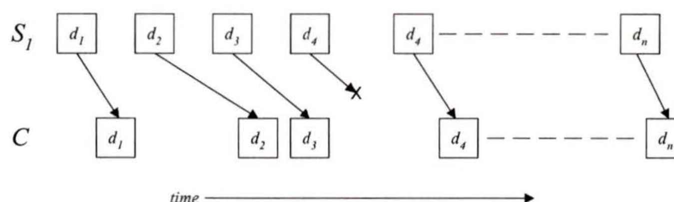


Figure 2.2: The transfer of discrete data

could simultaneously receive pieces of the discrete data from various surrogates and then put all the pieces back together to form the original discrete data.

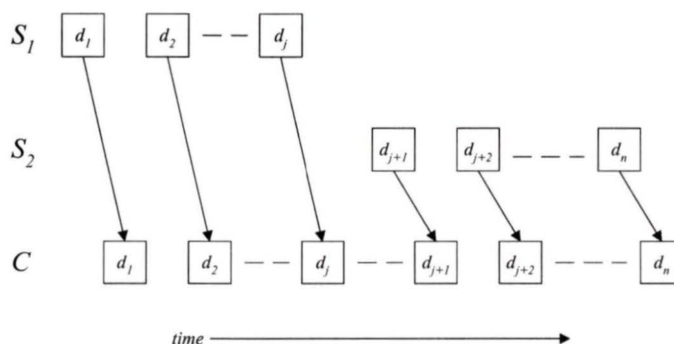


Figure 2.3: The transfer of discrete data from multiple surrogates

The discrete data is transferred from a surrogate to a client via an asynchronous *data* channel; messages are assumed to be ordered but unreliable. To recover from lost data messages, a sliding window or alternating bit protocol may be used [13]. Figure 2.2 shows the recovery of the lost message d_4. These protocols will slow down the performance and increase latency in the data transfer. This slow down is not an issue since the discrete data is not time sensitive.

A commercial discrete data delivery architecture similar to the one described above is Napster [18]. However, Napster has no notion of an origin. Each client except the Napster coordinator is a surrogate as well as a client for discrete data. The discovery and delivery of data in Napster uses the same four steps described on page 10 except that the client C picks the best surrogate S_j for itself from a list of surrogates provided by the coordinator.

2.3 Delivery of Continuous Data

Continuous data is time sensitive. It is delivered by the origin at a fixed periodic rate and must be received by the client at the same fixed periodic rate. Late data, possibly caused by network congestion, is treated as lost data. Moreover, retransmission of late data is meaningless; guarantee of data delivery is unnecessary as long as the loss rate is acceptable. If the continuous data is in real time, neither the origin nor any of the surrogates will have the future data. This means that a client cannot simultaneously receive future pieces of data from various surrogates like it could with discrete data.

The role of the origin and the surrogates are better defined for the delivery of continuous data than they were for the delivery of discrete data. The origin provides the original continuous data source for the surrogates. The surrogates act as proxies or relays of the continuous data for clients. The data channel communication shown in Figure 2.4 shows how data may be distributed in the generic continuous data delivery architecture. The control channel communication in the same figure shows the two way control channel connections between the elements. Every element has a control channel connection to the coordinator and, if receiving continuous data, a connection to the providing element. In this figure data originates at the origin, is distributed to the surrogates, and then to the clients. The coordinator does not participate in the data transfer but it coordinates the set up of all the data connections.

Figure 2.5 shows the transfer of the continuous data from the origin to the client. The fixed periodic rate of messages generated by the origin is the same fixed periodic rate of messages received by the client. Data passes through an intermediate layer before arriving at the client. The intermediate layer may be single or multiple layers of relaying surrogates. Any minor jitter experienced from the client may be eliminated through the use of buffers at the client. Some data message loss may be acceptable if the continuous data stream has some form of forward error correction (FEC) [3]. Missing data messages can be interpolated using FEC; this is shown with message d_3 in Figure 2.5. Reed Solomon codes [19] are one example of FEC codes.

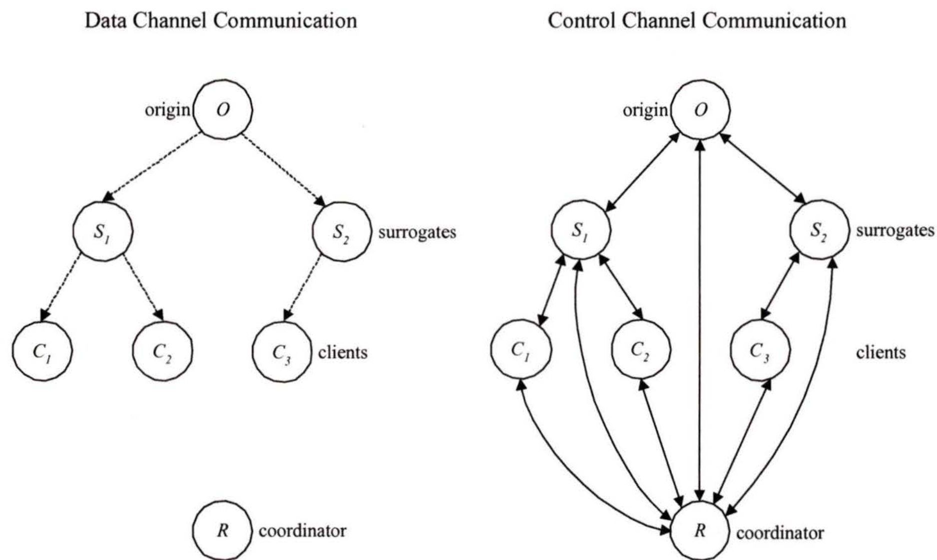


Figure 2.4: Data channel and control channel connections

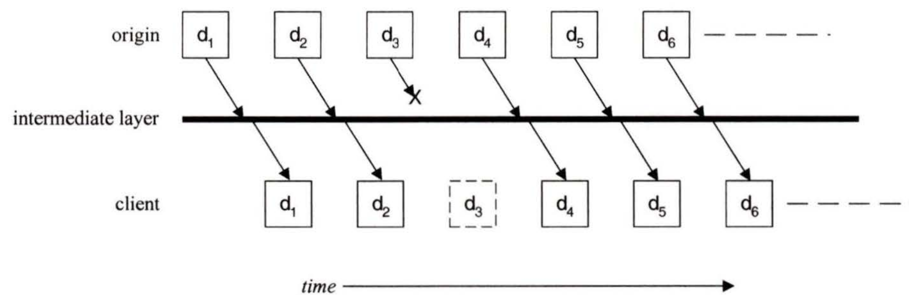


Figure 2.5: The transfer of continuous data from the origin to client

Receiving continuous data from multiple surrogates is considerably more interesting than receiving discrete data from multiple surrogates. Due to varying distances from the client to the surrogates, surrogates will be at different latencies from each other. Also migrating from one stream to another must not cause noticeable jitter in the data stream. These issues are the topics of the next chapter.

Chapter 3

The Client Migration Protocol

This chapter describes the design of the client migration protocol. The description of the protocol uses the generic content delivery architecture presented in Chapter 2. A sliding window is used in the design of the migration protocol. The first part of this chapter introduces the sliding window and how migration is limited by the size of the window. Next the migration protocol is presented. This is followed by a discussion of various faults that may occur and how these faults are handled.

The design of the protocol can be found in Appendix A. The appendix describes all the messages and state transitions for each of the network elements involved in migration. These elements include the coordinator, the surrogate, and the client.

3.1 A Sliding Window

The delivery of continuous data requires the periodic timely delivery of data messages from a producer to a consumer. The consumer consumes these data messages at the same periodic rate as they are produced. Late data is considered the same as lost data. The timely delivery of continuous data messages is difficult on the Internet due to unpredictable congestion. Congestion introduces jitter and delay that, as a result, causes lost data. One method for dejittering a data stream is to use a buffer. On initial connection to the producer, a buffer is partially filled with data messages prior to their being consumed. Therefore

when a data message arrives late there are some data messages in the buffer available for consumption. However, consecutive late data messages will cause the buffer to empty and lead to a buffer underrun. If a buffer is too small, a burst of data messages may cause the buffer to overrun.

We model a buffer with a sliding window. A sliding window is a fixed size array that encapsulates a range of M data messages. All data messages are of the same uniform size. Figure 3.1 shows a window of size $M = 6$ with cells B_0 to B_5 covering part of a continuous data stream. Each number in the continuous data stream represents the sequence number of a data message. The window has a read head labelled r and a write head labelled w. The values of the read head and the write head specify their relative location within the window. The read head r always has the value 0. The write head w has a value between 0 and M. The write head sits at the next empty location in the window except for when w is equal to M. The data stream messages to the left of B_r have been consumed. If $w > 0$, then messages held in the cells B_r to B_{w-1} are to be consumed. The remaining cells are empty.

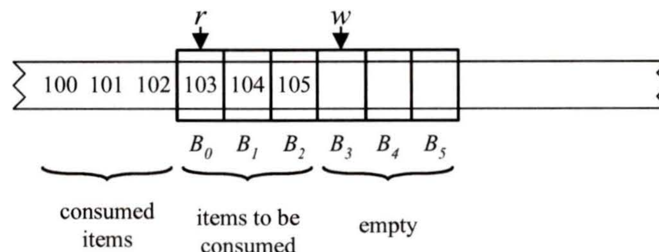
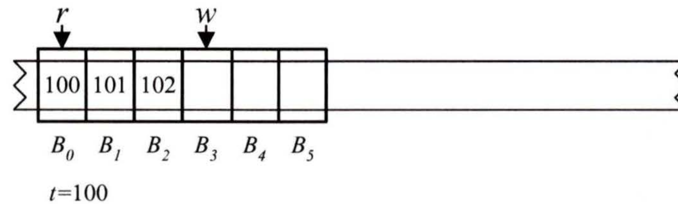


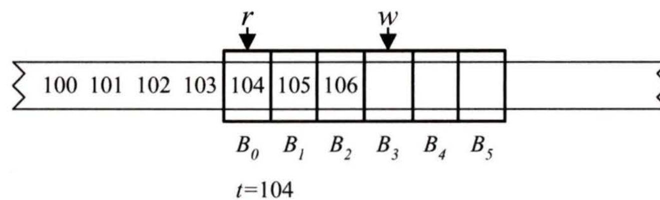
Figure 3.1: A sliding window B with a read and write head

Assuming the data stream is ordered and reliable, we maintain a time counter t to specify the absolute position of the beginning of the window within the data stream. Figure 3.2 shows the window at time $t = 100$ and at time $t = 104$. In general, at time $t = k$, $B_0 = k, ..., B_i = (k + i)$ for all $i < w$, and $B_{k+j} = empty$ for all j where $w \leq j < M$.

The counter t is incremented after every *read* operation and the window is slid to the right by 1. When the window slides to the right and $w > 0$, w is implicitly decremented by 1 to remain at the same absolute position within the data stream. Figure 3.3 shows this



(a) At time $t = 100$, $B_0 = 100$, $B_1 = 101$, $B_2 = 102$, $B_3 = B_4 = B_5 = empty$



(b) At time $t = 104$, $B_0 = 104$, $B_1 = 105$, $B_2 = 106$, $B_3 = B_4 = B_5 = empty$

Figure 3.2: The sliding window at time $t = 100$ and at time $t = 104$

change in t and w after a data message is read. When w is between 0 and $(M - 1)$ an incoming data message is written to B_w and w is incremented by 1. Figure 3.3 shows this change in w can be seen after a data message is written. A window is underrun if $w = 0$ when reading. A window is overrun if $w = M$ when writing. Underruns and overruns of the window cause discontinuities in the data stream.

Figure 3.4 shows the use of sliding windows for each network element of a continuous content delivery network discussed in the previous chapter. An origin delivers data to two surrogates and then each surrogate delivers data to a client. For simple illustration, the origins and surrogates have a window size of two and the clients have a window size of six. In a real network, the buffer size of the origins, surrogates, and clients will be larger. The data messages that are in transit between the elements in Figure 3.4 illustrate network latency. Surrogates may have different network latencies in their data paths to the origin.

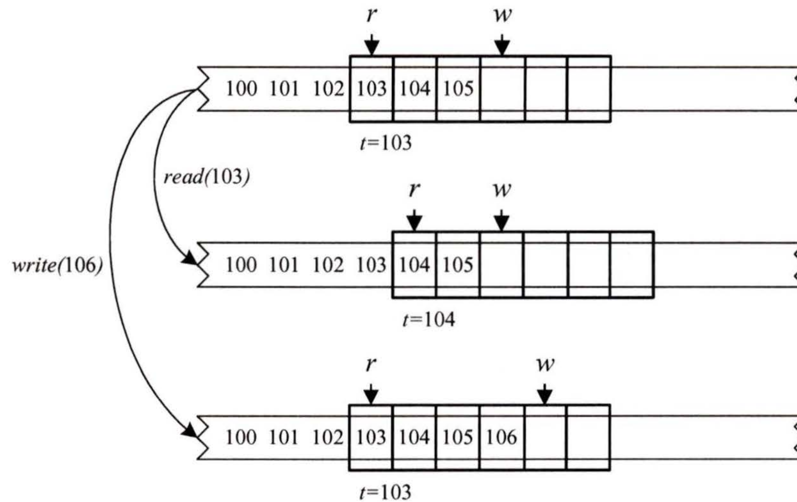


Figure 3.3: The actions of a read operation and a write operation on a sliding window

This latency difference can be seen by the different values of time t for the surrogates S_1 and S_2. Surrogate S_1 has a higher value of time t than surrogate S_2 meaning that S_1 has less latency to the origin O. Latency differences are further propagated to the clients. Client C_1's value of time t is greater than client C_2's value of t.

3.2 Client Migration

The main purpose of client migration is to overcome the jitter or data discontinuities between a client and its surrogate. Client migration allows a client to migrate to another surrogate when the client detects that its current data stream is experiencing jitter or degradation of service. The migration from the existing surrogate stream to a new surrogate stream requires the splicing of the two streams to form a continuous stream. As discussed previously surrogates may have different time t values due to network latencies. The stream splicing algorithm must take latency differences into account and splice the two streams to preserve the monotonically increasing order of the sequence numbers without introducing discontinuity. In order for a migration to succeed the following three steps must take place:

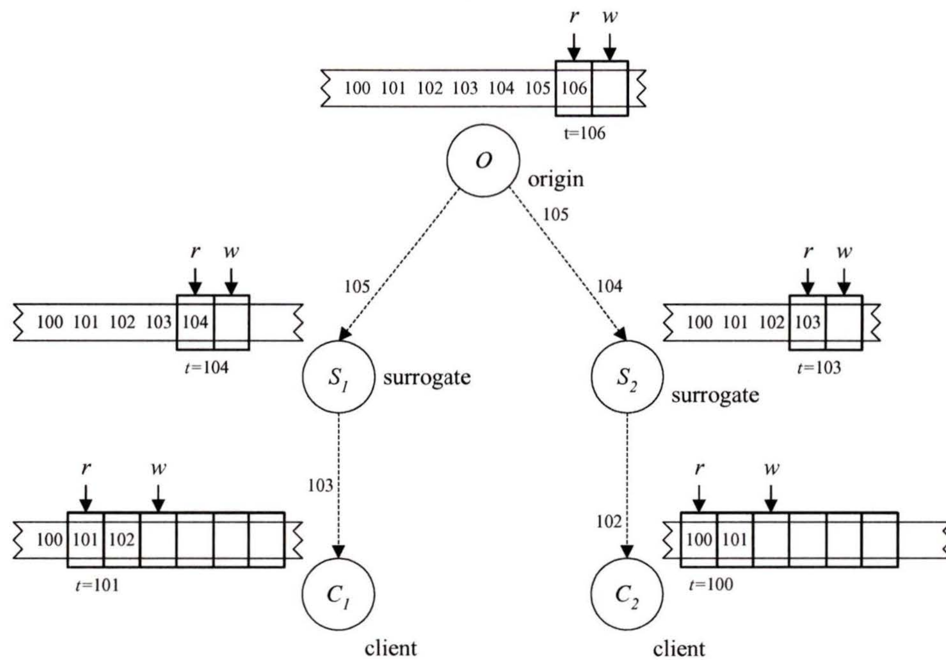


Figure 3.4: Data channel communication in a continuous data network

1. an acceptance test - *whether* a new data stream is acceptable;

2. the splice point check - *where* to splice the new data stream; and

3. a continuity test - *when* the old data stream is finally spliced with the new data stream.

The acceptance test determines if a new data stream fits within the bounds of the client's current window. A client is limited to the surrogates that it is able to migrate to by the following properties: the client's window size M, the client's value of time t, and the new surrogate's value of time t.

Figure 3.5 shows a client and five surrogates that have different values of t. The client C is currently receiving data from surrogate S_1 and has a t value of 103. Examination of the figure shows that C may only migrate to surrogates S_2 and S_3. The time t values of surrogate S_2 and S_3 fit within the bounds of C's window. Migration to surrogates S_4 and S_5 is impossible since the t values of each of these surrogates are out of the bounds of the

client's window. In general the migration of client with time $t = k$ to a new surrogate with time $t = x$ is only acceptable if $(x - k)$ is between 0 and $M - 1$. The t value of a surrogate is determined by the sequence number of the first data message from the new surrogate's stream. If the stream is accepted, the splice point becomes $s = (x - k)$.

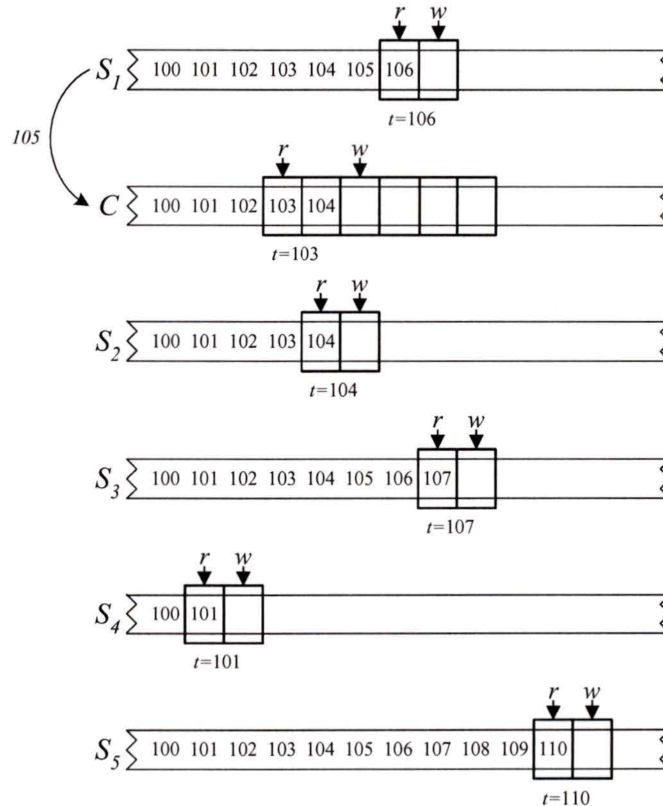


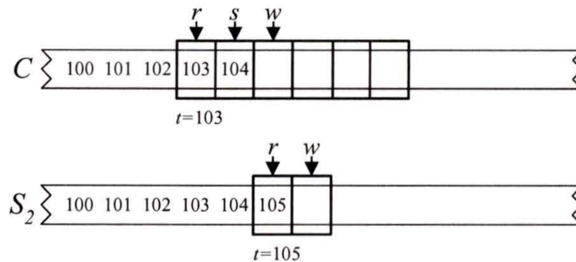
Figure 3.5: Surrogates and a client at various positions in a data stream

After a successful acceptance test, a decision must be made on where to begin saving the new data stream in the window. This is referred to as the splice point check. The value of the splice point specifies its relative position within the window. Using the splice point $s = (x - k)$, discussed in the previous paragraph, there are two cases for the relation between s and the client's write head w:

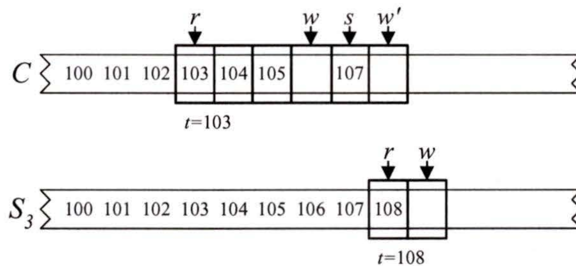
1. $s \leq w$ - the splice point s is *at* or *earlier* than the write head w; and

2. $s > w$ - the splice point s is *later* than the write head w.

Both of these cases are illustrated in Figure 3.6 by showing the placement of the splice points for client C's migration to S_2 and S_3 of Figure 3.5. Both S_2's and S_3's windows have advanced within the data stream since each surrogate has sent an initial data message. The initial data message is written to cell B_s.



(a) A splice point s is *earlier* than the client's write head w



(b) A splice point s is *later* than the client's write head w

Figure 3.6: Client C's migration to the valid surrogates S_2 and S_3 of Figure 3.5

Once the splice point is chosen a final test for data stream continuity must be performed. A client's data stream becomes continuous when $w \geq s$; this is known as a successful splice. For the first splice point case where $s \leq w$ the data stream is automatically continuous, the messages in the window from $B_{(s+1)}$ to B_w are discarded, and the write head w is reset to $s + 1$. For the second splice point case where $s > w$ there is a gap between B_w and B_s meaning that the data stream in the window is not continuous. A write

head for the new stream labelled w' is situated at B_{s+1}. The stream is not continuous until enough data messages arrive from the existing surrogate so that $w = s$. When the stream finally becomes continuous the write head w is reset to w'.

3.3 Protocol Design

The client migration protocol consists of messaging and stream splicing. The messaging required to perform the migration occurs on the *control* channels of the following network elements: the client, the coordinator, the client's existing surrogate, and the new surrogate to be hosting a data stream to the client. Stream splicing, as discussed previously, is concerned with the splicing of the existing surrogate's data stream and the new surrogate's data stream that are both delivered on the *data* channel. The *control* channel and the *data* channel are orthogonal; i.e. the messages of either channel are independent from each other.

For the protocol design we assume that *control* channel communication is unreliable. Ordered communication for the control channel is not important since the messaging is ordered according to the state of the migration. The *data* channel communication is assumed to be ordered and reliable. Later in the Fault Analysis section we discuss how the protocol may handle unreliable but ordered *data* channel communication.

The origin produces continuous data messages at a fixed frequency f. These continuous data messages are then delivered to the surrogates and then to the clients at the same frequency f. On initial connection the client stores a set amount of data in its data buffer before consuming it at the frequency f. The number of items in the client's data buffer may fluctuate due to jitter in the network. If the network was absent of jitter the buffer would always contain a constant number of items since data messages arrive at the same rate as they are consumed.

Figure 3.7 shows a message sequence chart for the client migration protocol. Both control and data channel communication are shown. In this figure the coordinator R initiates the migration of the client C from its existing surrogate S_1 to the new surrogate S_2.

The coordinator R chooses the new surrogate S_2 as the *best* alternate surrogate for C.

As discussed in the previous chapter, the definition of *best* may be based on one or more of the following properties: the geographic location of a surrogate with respect to a client, the network locale of a surrogate and a client, the number of hops from the client to a surrogate, the current network resources of a surrogate, the current computing resources of a surrogate, or the quality of the *data* channel between a surrogate and the client. The primary objective of choosing the *best* alternative surrogate is to optimize the client's QoS.

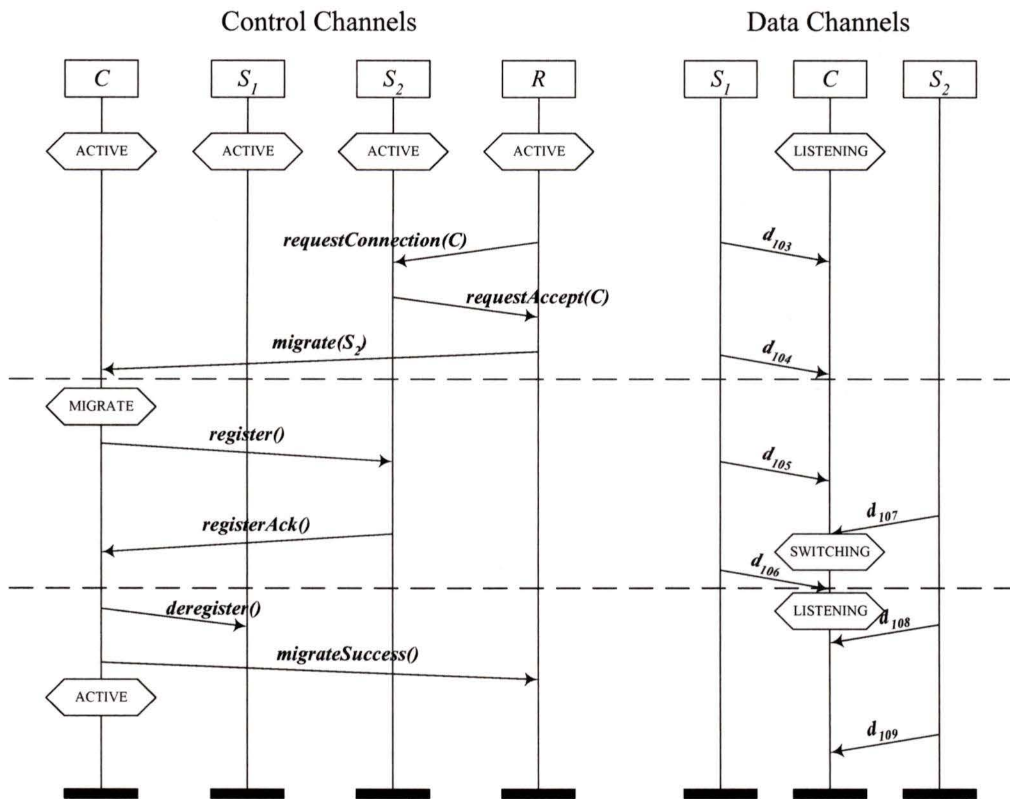


Figure 3.7: The message sequence chart for a client migration

Before R was able to initiate the migration it had to determine if S_2 could support another stream connection by sending the message ***requestConnection(C)*** to S_2. The parameter C was the ID of client C. This ID was used by S_2 to allow registration from C. Upon acceptance of the request, S_2 responded to R with the message ***requestAccept(C)***.

Next, R sent the message ***migrate(S_2)*** to C. The parameter S_2 was the ID of new surrogate S_2. At this point C's control channel changed from the *ACTIVE* state to the *MIGRATE* state. The first dashed line shows the start of migration for the client. The change in state allowed the data channel to accept messages from S_2. To start receiving the data messages from S_2, C registered to S_2 by sending the message ***register()*** to S_2. Upon receiving the ***register()*** message, S_2 immediately started sending data to C. To notify C of a successful registration S_2 replied with the message ***registerAck()***.

Upon receiving the first data message from S_2, C's data channel changed from the *LISTENING* state to the *SWITCHING* state. As discussed previously, the sequence number of the first data message received from S_2 is the time t of S_2. The value of time t passed the acceptance test, and the first data message was written to the splice point shown in Figure 3.8. The data stream inside the window did not become continuous until the message d_{106} arrived from S_1. The second dashed line in the message sequence chart shows the completion of the migration for the client. At this point the write head w was reset to w' and the data channel changed back to the *LISTENING* state. C now only accepted data messages from S_2; any additional data messages from S_1 were ignored.

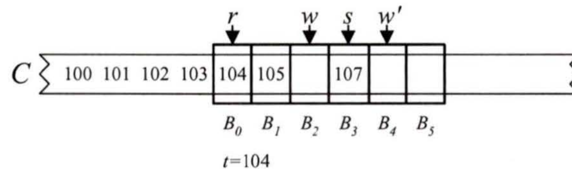


Figure 3.8: The window showing the splicing of the two streams

After a successful splice, C deregistered from S_1 by sending the message ***deregister()***. S_1 immediately stopped sending data to C. To complete the migration C sent the message ***migrateSuccess()*** to R and its control channel returned to the *ACTIVE* state.

Figure 3.9 shows what happened when the time t value from S_2 failed C's acceptance test. The t value of S_2 was sequenced too far upstream from C's write head to fit within the bounds of C's buffer. After failing the acceptance test, C sent a ***deregister()*** message to S_2, and a ***migrateInvalid()*** message to the coordinator R. C's control channel

returned to the *ACTIVE* state, and C continued to receive data from S_1. After receiving the ***migrateInvalid()*** message, the coordinator R could initiate another migration to a different surrogate.

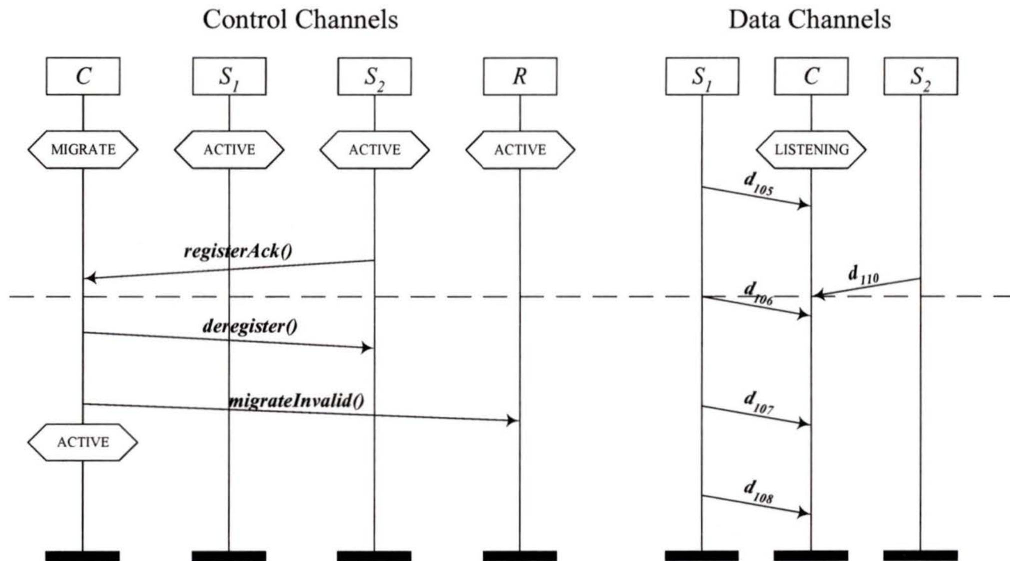


Figure 3.9: Client migration to a surrogate too far upstream from the current stream

3.4 Fault Analysis

Client migration is necessary to transfer a client from a surrogate of poor QoS to a surrogate with better QoS. An existing surrogate with poor QoS may cause control or data message faults during migration. The faults may cause discontinuities in the increasing monotonic order of the messages within the window. Up to this point it has been assumed that the data stream was continuous and had the property such that the datum to the right of d_x was always d_{x+1}, where the subscript represented the sequence number. For this section we loosen this requirement and require that the sequence numbers in the stream arrive in strictly increasing order.

The continuous requirement of the buffer means that the data stream inside the win-

dow must contain no gaps. Gaps may occur when there is a missing message in the existing stream during the splicing of a data stream *later* in sequence from the existing stream. A gap can be removed by discarding the cells that make up the gap. Figure 3.10 shows the removal of the gap after a lost message in the existing stream. Once the gap is removed, the write head w is reset to w' and the new data stream continues to fill the buffer.

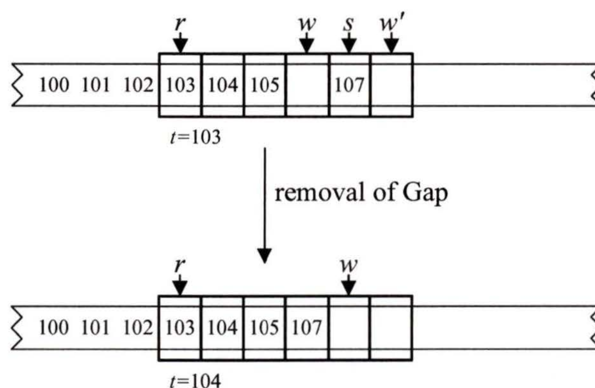


Figure 3.10: The removal of a gap after a failed migration to a *later* stream

Strictly increasing means that the data message to the left of d_x is always d_{x+y} where $y \geq 1$. Figure 3.11 shows a window with sequenced messages in strictly increasing order. Notice how message 105 has been lost and that message 106 has been written in the cell where message 105 should have been. It is up to the consumer of the window's data messages to handle the discontinuities in sequence numbers through the use of error correction or other means. The consumer must adjust its *get* frequency on the buffer according to missing sequenced messages.

The strictly increasing property means that the sequence number of the message at B_r may not be equal to time value t. To find the value t we use the sequence number of the datum in B_{w-1} if $w > 0$. The value of t is then $B_{w-1} - (w-1) = B_{w-1} - w + 1$. If $w = 0$ then the value of t is the last consumed sequence number incremented by 1. The t value in Figure 3.11 is not the same as the sequence number in cell B_r. This value of t ensures that the splice point always inserts the new stream such that the strictly increasing order

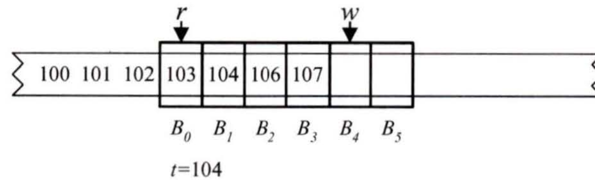


Figure 3.11: Non-monotonically increasing sequence numbers of the data stream

is preserved. For example if the first data item of the new stream has a sequence number 106, for the window in Figure 3.11, using a t value of 103 from $t = B_r$ would place it at cell B_3. This would cause violation of the strictly increasing property of the stream since 106 already exists at B_2. However if we calculate t in reference to B_{w-1} the datum with sequence number 106 would be placed in cell B_2. A splice point will always preserve the strictly increasing order since the t value calculated in reference to B_{w-1} assumes a monotonically decreasing stream going left from B_w.

3.4.1 Control Message Faults

Control message faults include lost control messages, duplicate surrogate migration requests, or logout from either the existing or new surrogates during migration.

3.4.1.1 Lost Control Messages

Timers provide detection for lost messages. Figure 3.12 shows the message sequence chart for a client migration with the use of timers. The entities labelled with T are the timers: an hourglass symbol represents the creation of a timer whereas a cross symbol represents the destruction of a timer.

The timer on R may timeout from lost control messages with S_2 or C and results in a failed migration. The timer on S_2 may timeout from lost control messages with R or C and results in the cancellation of the stream registration.

After the ***migrate(S_2)*** message from R, the client C sets a timer to ensure that migration will complete. This timer will timeout if no messages are received from the new

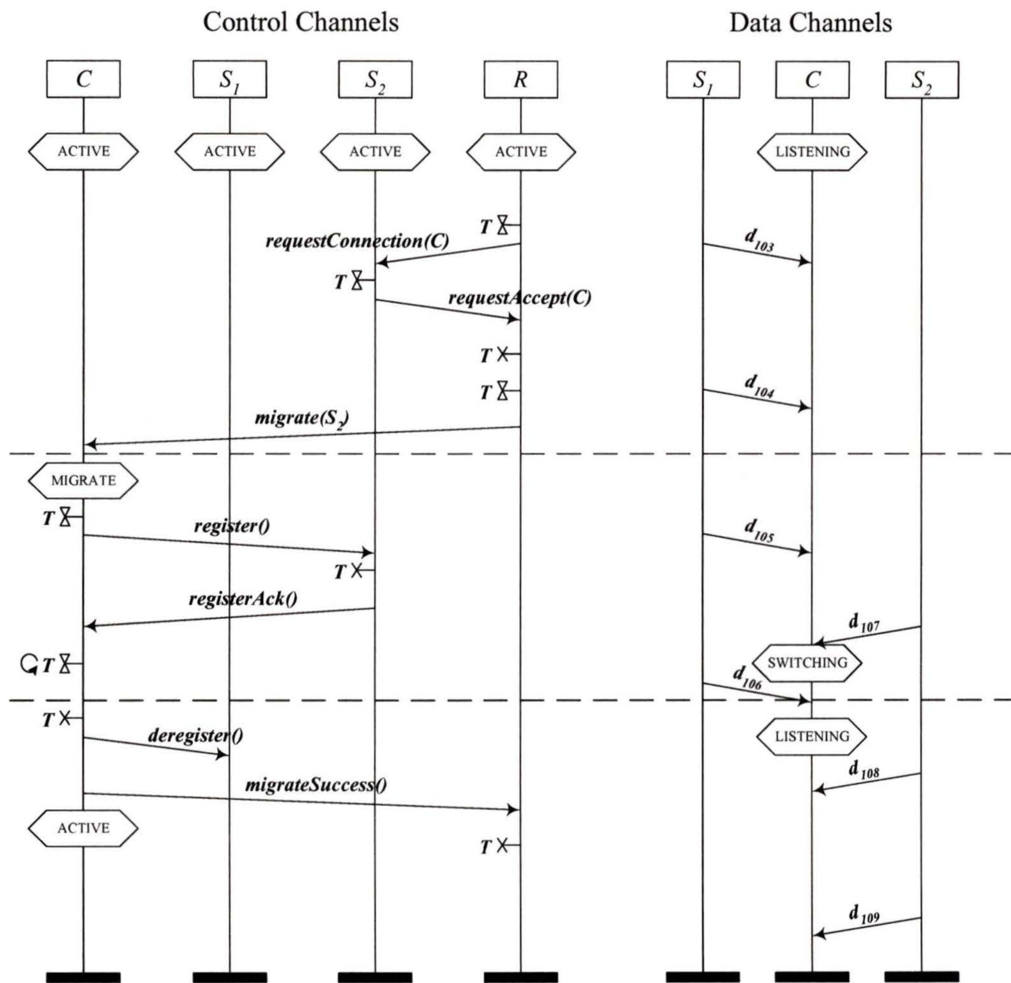


Figure 3.12: The message sequence chart for a client migration with timers

surrogate. A timeout will cause C to return to the *ACTIVE* state, send a ***deregister()*** message to S_2, send a ***migrateFail()*** message to R, and continue to receive data from S_1.

When the first data message arrives from S_2, the splice point is determined from the sequence number. If the splice point is at the same position or at an earlier position relative to the write head then migration will complete normally and the timer will be discarded. Otherwise if the splice point is in a later position the data channel state changes to the *SWITCHING* state and the timer is reset for how long it should take the read head to reach

the splice point. This time is described by the equation $T \times (s - r)$ where T is the period of the data. This is the maximum time the client can wait for the existing surrogate to overlap with the new stream. If an overrun occurs before the migration is complete empty data is returned. An overrun means that older data from the existing surrogate may still arrive. This older data is simply discarded upon arrival. If the timer times out before migration is complete then r will become s, w will become w', and the data channel will change back to the *LISTENING* state. Next, C will return to the *ACTIVE* state, send a ***deregister()*** message to S_1, send a ***migrateSuccess()*** message to R, and receive data from the new surrogate.

To ensure that a connection to a client is still active a surrogate periodically sends a round trip ***heartbeat()*** message on the control channel. If the ***deregister()*** message from C to S_1 becomes lost, S_1 will stop receiving responses to the ***heartbeat()*** messages and will eventually close the stream connection.

3.4.1.2 Duplicate Surrogate Migration Requests

A fault occurs if S_2 in the ***migrate(S_2)*** message from the coordinator contains the ID of the client's existing surrogate. Since it does not make sense for the client to migrate to a surrogate it is already connected to, the client will respond to the coordinator with the ***migrateDuplicate()*** message. Upon receiving the ***migrateDuplicate()*** message R may initiate another migration. A duplicate surrogate request may arise from a delayed migration request from a previous surrogate that hosted the client.

3.4.1.3 Logout from a Surrogate During Migration

The existing or new surrogates may notify the client of a stream shutdown during the client's migration. Actions are taken depending on how far the client is through migration.

There are two cases to handle if the existing surrogate notifies the client of a stream shutdown. The first case occurs when the data channel is in the *LISTENING* state. This means that the client has not received the first data message from the new surrogate. If this happens the client will do nothing and wait for its timer to timeout or a data message to

arrive from the new surrogate. If the migration completes due to a timeout, a notification of stream shutdown from the new surrogate, or data out of range, then the client will assume it does not have any stream connections and will terminate. If a data message containing a sequence number that passes the acceptance test, does arrive from the new surrogate before timeout, any gap between the existing stream and the new stream will be removed as shown in Figure 3.10, and the migration will complete. The completion of the migration will cause the client to return to the *ACTIVE* state, send a ***migrateSuccess()*** message to R, and receive data from the new surrogate.

The second case for the notification of stream shutdown from the existing surrogate occurs when the data channel is in the *SWITCHING* state. This means that the client is receiving data from the new surrogate. If the existing surrogate notifies the client of stream shutdown then a migration will be forced. Any gap between the existing stream and the new stream will be removed as shown in Figure 3.10. A forced migration will cause the client to return to the *ACTIVE* state, send a ***migrateSuccess()*** message to R, and receive data from the new surrogate.

If the new surrogate notifies the client of a stream shutdown during migration before the ***deregister()*** message has been sent to the existing surrogate, the client can just continue receiving from the existing surrogate, and send a ***migrateFail()*** message to the coordinator. Otherwise if the ***deregister()*** message has already been sent to the existing surrogate the client will have to start a new connection.

3.4.2 Data Message Faults

Data message faults include messages that are late or lost. As discussed previously late messages are caused by network jitter. The solution to solving minor jitter was to use a buffer. Consecutive lost messages may be due to the failure of the stream or failure of the surrogate serving the stream. If we continue to write every incoming data message to the next empty cell, provided that space is available, then the window will keep the required continuous and strictly increasing sequence number order in the presence of lost messages.

Using the method discussed previously for calculating t on a non-monotonic se-

quenced stream, the acceptance test and the splice point check steps of client migration remain unchanged. If $s \leq w$ then the data continuity test will pass and migration is complete. However, the data continuity test step must be changed when $s > w$. When $s > w$ three conditions can be used to test for whether a data stream is continuous in the presence of lost messages:

1. test for $w = s$;
2. test for $B_w \geq B_s$; and
3. reset the client's timer for $T \times (s - r)$.

The first test for continuity becomes true when there are no lost messages in the existing stream during migration. The second test becomes true when there are lost messages on the existing stream during migration. Similar to the situation shown in Figure 3.10 the gap is removed and the write head w is reset to the new stream's write head w'. The third test, discussed previously, relates to the timer set by the client at the beginning of migration.

The quality of the new stream is difficult to determine. The coordinator's measure of *best* may not lead to a better QoS stream for the client. The time taken for the data stream within the window to become continuous may not be enough to determine the QoS of the new stream. Migration may need to be requested over and over before a stream with an acceptable QoS is found.

3.4.2.1 Buffer Underruns and Overruns

During client migration buffer underruns and overruns may occur. Various actions may be taken for buffer underruns and overruns. These actions are engineering specific and depend on the type of the continuous data that is transferred and the quality requirements of the continuous data.

There are two possible choices to make when a buffer underrun occurs. The first choice is for the client to keep receiving from the existing stream until the migration timer times out. When the migration timer times out the client is forced to go to the new stream.

If the client keeps with the existing stream, the client has to decide what to return to the consumer of the data buffer. Depending on the parameters for the consumer of the data buffer, the client could return empty data, or an error message. The advantage of keeping with the existing stream until timeout is that a burst of data may arrive from the existing stream. The disadvantage of keeping with the existing stream is that no additional data may arrive from the existing stream.

The second choice for buffer underrun is for the client to immediately switch to the new stream. If the first message has arrived from the new stream the client's read head r is set to the splice point, the write head w is set to the write head of the new stream w', and the message at the splice point is sent to the consumer of the data buffer. The advantage of going to the new stream is that the consumer receives data immediately. The disadvantage of going to the new stream is that a large discontinuity could be introduced depending on the distance between the original read head r and the splice point s within the buffer. Also data may not have yet arrived from the new stream or the new stream may be invalid causing failure of the client.

A buffer overrun may occur during migration due to a burst of data from the new stream. There are two possible choices for the data if a buffer overrun occurs. The first choice is to discard any new data when the new stream's write head w' is at the end of the buffer. This choice will cause a discontinuity later on in the consumption of the stream. The second choice is to advance the read head r in order to write the new data. This choice will cause an immediate discontinuity for the consumer of the data stream.

Chapter 4

Experimental Validation

This chapter describes the validation of the client migration protocol. Various experiments demonstrate the distributed initiation, idempotency, and resiliency properties of the protocol. Finally, a performance evaluation of the protocol is given.

4.1 Experimental Framework

The client migration protocol experiments were implemented in the coordination language COOL [20]. COOL is an object based language that facilitates the creation of distributed applications. COOL's code is compiled to C/C++ and works with the runtime system STEAM. COOL is composed of objects and timers. Objects may run on one computer or across multiple computers interconnected with a data network. Communication between the objects occurs via asynchronous messages that are ordered and unreliable. COOL programs may be integrated with other C libraries.

The experiments discussed in this chapter were executed on Pentium level computers running a combination of Windows and Linux. The computers were interconnected in a 10Mbps ethernet network; COOL used UDP/IP for communication between computers. Experiments were run stand alone on a single computer, and distributed across multiple computers.

Figure 4.1 shows a generic data delivery application as it appears in COOL. The

dotted square outline represents each network element and the circles represent COOL objects. A control channel is represented by a circle labelled "cc" and a data channel is represented by a circle labelled "dc". For each of the experiments there is one coordinator, one origin, one client, and two or more surrogates labelled R, O, C, and S_x respectively. The subscript x for the surrogate represents the surrogate number. The experiments are run as stand alone or distributed. In the stand alone case all elements are run on one computer. In the distributed case each of the network elements run on their own computer and communicate with each other across the network. The dotted lines connecting each of the data channel objects show the propagation of continuous data through the network. The two way solid lines connecting the control channel objects show the control channel connections. The control channel is dominant and has control over the data channel in each element. Each experiment begins with client C receiving continuous data from surrogate S_1.

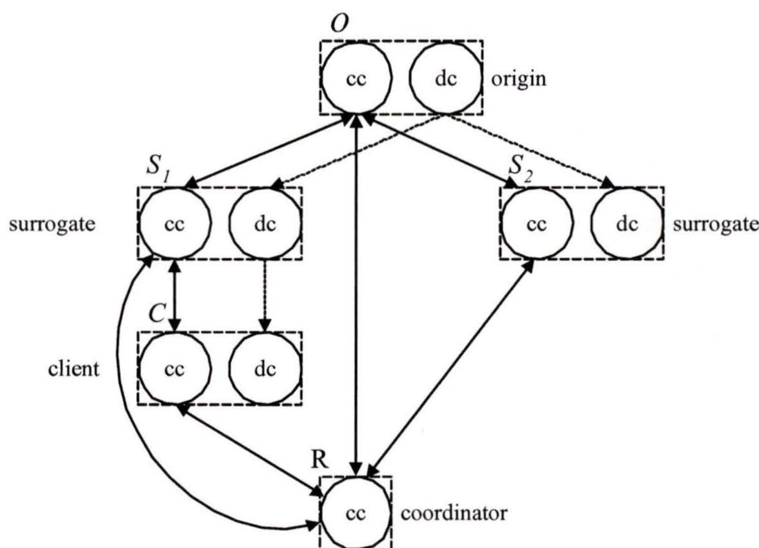


Figure 4.1: The Generic Data Delivery Application as it appears in COOL

The main goal of the client migration protocol is to migrate the client from one surrogate to another without an interruption or duplication in the data stream. Figure 4.2 shows

that the fixed rate of data released from the origin is the same fixed rate received by the client. The intermediate layer may consist of one or more surrogates. It is imperative that the migration preserves the fixed rate of continuous data, preserves the strictly increasing message order, and loses no messages. All the experiments fulfilled this requirement.

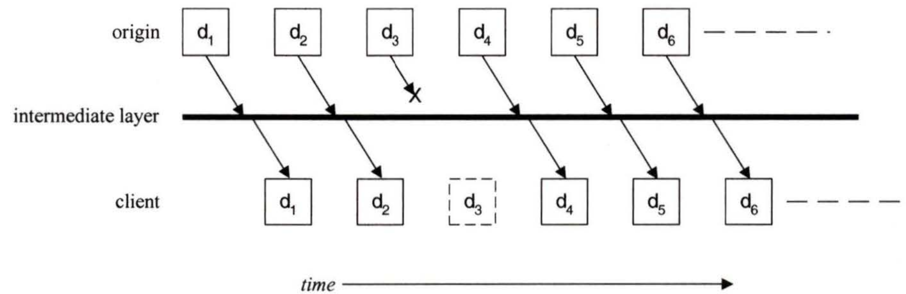


Figure 4.2: The transfer of continuous data from the origin to the client

4.2 Distributed Initiation

Distributed initiation is the ability of any entity, except the origin, to initiate migration. A coordinator may initiate a client to migrate after monitoring network load. A server may initiate migration if it is overloaded. A client could detect a deterioration of the quality of its data stream and initiate migration. Due to the nature of the client migration protocol, multiple requests for migration may occur simultaneously.

Experiments demonstrate that a client, surrogate, or coordinator could manually initiate a client migration. To initiate a client migration the client or surrogate could send a ***requestMigrate(CId, SId)*** message to the coordinator. The argument CId is the ID of the client that is to be migrated and the argument SId is the ID of the client's existing surrogate connection. A coordinator could manually initiate a client migration by sending an internal ***requestMigrate(CId, SId)*** message to itself. Each entity has its own reason for initiating migration. A client initiates migration to maintain data continuity. A surrogate initiates migration to manage its load. A coordinator initiates migration to manage network faults, and maintain efficient utilization of network resources.

Once it was demonstrated that distributed initiation was possible, an experiment to show an automatic migration was conducted. An automatic migration would occur when the client experienced jitter on its current stream. To demonstrate the automatic migration a jitter detection mechanism was added to the client. The experiment first started by the client receiving continuous data from one surrogate. A load was initiated at the surrogate to introduce jitter in the data stream. Immediately the client detected the jitter, requested migration, and was successfully migrated to another surrogate. Automatic migration showed that the client migration protocol could be adaptive.

4.3 Idempotency

An important property when using an asynchronous message based network is idempotency. An operation is idempotent if multiple initiations of the same request result in a single request. It is important that the migration of a client from one surrogate to another is idempotent. For example, if a client is currently migrating, any additional requests to migrate are ignored. Idempotency must occur at the coordinator and at the client. Idempotency at the coordinator is necessary to handle multiple migration requests from a surrogate and a client during the migration of a client. Idempotency at the client is necessary to handle multiple migration requests from the coordinator during and after a client's migration. Multiple requests may come from the coordinator if the coordinator does not receive a response from the client that a migration had occurred.

Figure 4.3 shows how the coordinator R handled multiple requests for migration from the client C and the surrogate S_1. The first migration request was from C and the migration began immediately. A short time later R received the migration request from S_1. C was still migrating so R just ignored the second migration request.

A similar experiment demonstrated how the client handled multiple requests for migration from the coordinator during migration. The experiment demonstrated that the client did not even accept a second ***migrate()*** message from the coordinator while in the *MIGRATE* state.

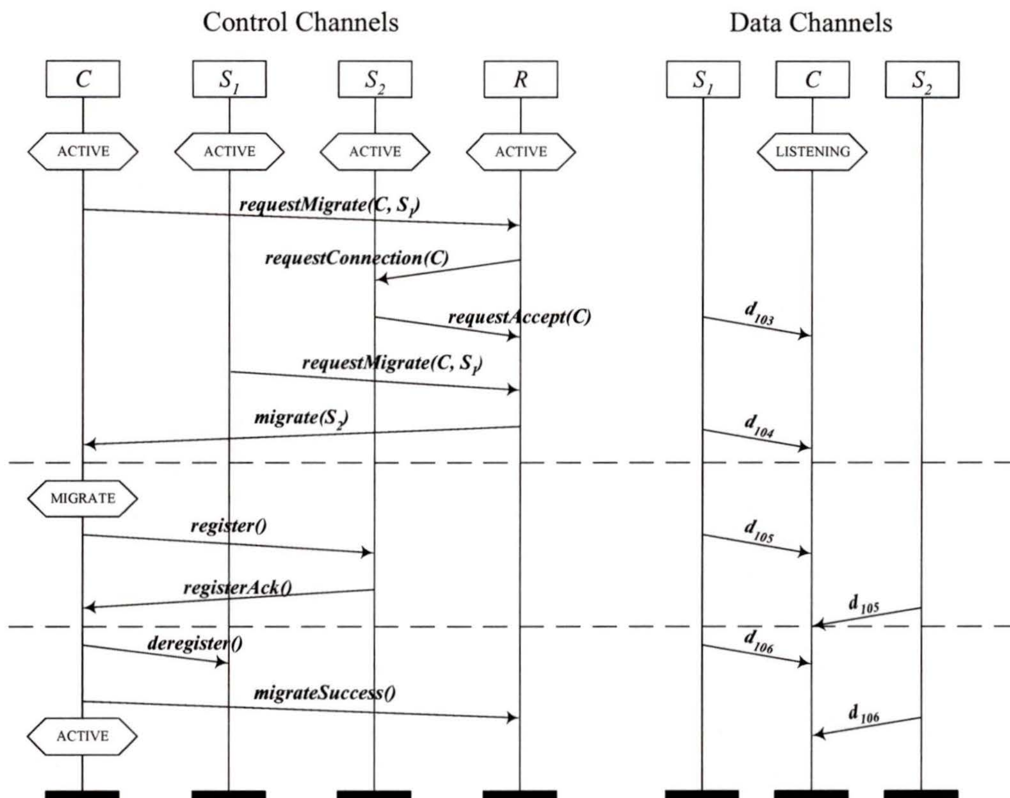


Figure 4.3: The message sequence chart showing multiple migration requests from a client and surrogate during a single client migration

To demonstrate the handling of a duplicate message after migration completed, an experiment showed that a request for migration to the same surrogate would be ignored. Upon receiving the duplicate migration request the client responded to the coordinator with the message ***migrateDuplicate()***.

4.4 Resiliency

Resiliency of the migration protocol is the ability to handle varying surrogate latencies, control message faults, and some data stream faults. Experiments demonstrate each of these resiliency conditions.

4.4.1 Varying Surrogate Latencies

Two experiments test the correctness of the migration protocol handling varying surrogate latencies. The experiments were migration to a surrogate downstream from the existing surrogate and migration to a surrogate upstream from the existing surrogate. Latency was introduced to a surrogate by the use of a buffer; each unit of latency corresponded directly to one unit in the buffer. A latency of ten data messages would require a buffer size of ten units.

Figure 4.4 shows the migration of the client C to the surrogate S_2 from S_1. S_2 is delivering data at a latency of two data units downstream from S_1. C initiates migration. The first six control messages handle the set up of the data stream from S_2. Once the second data stream is set up, C concurrently receives data from two data streams. In the figure the first message to arrive from S_2 after the second data stream set up is d_{103}. Since this message sets the splice point earlier than the write head the buffer becomes continuous immediately. Next, C sends a message to cancel the stream from S_1, and sends a message to notify R of a successful migration. As soon as S_1 receives the ***deregister()*** message from C it stops sending data to C. Any messages from S_1 sent to C after the buffer has become continuous are ignored.

Figure 4.5 shows the migration of the client C to the surrogate S_2 from the surrogate S_1. S_2 is delivering data at a latency of two data units upstream from S_1. Similar to the previous example, C initiates migration. The first six control messages handle the set up of the data stream from S_2. Once the second data stream is set up C concurrently receives data from two data streams. In the figure, the first message from S_2 to arrive after the second stream set up is d_{107}. This sets a splice point two message units ahead of the write head. The message d_{106} from S_1 makes the buffer continuous. Next, C sends a message to S_1 to cancel the stream and a message to notify the coordinator R of a successful migration. As soon as S_1 receives the ***deregister()*** message from C it stops sending data to C. Any messages from S_1 sent to C after the buffer has become continuous are ignored.

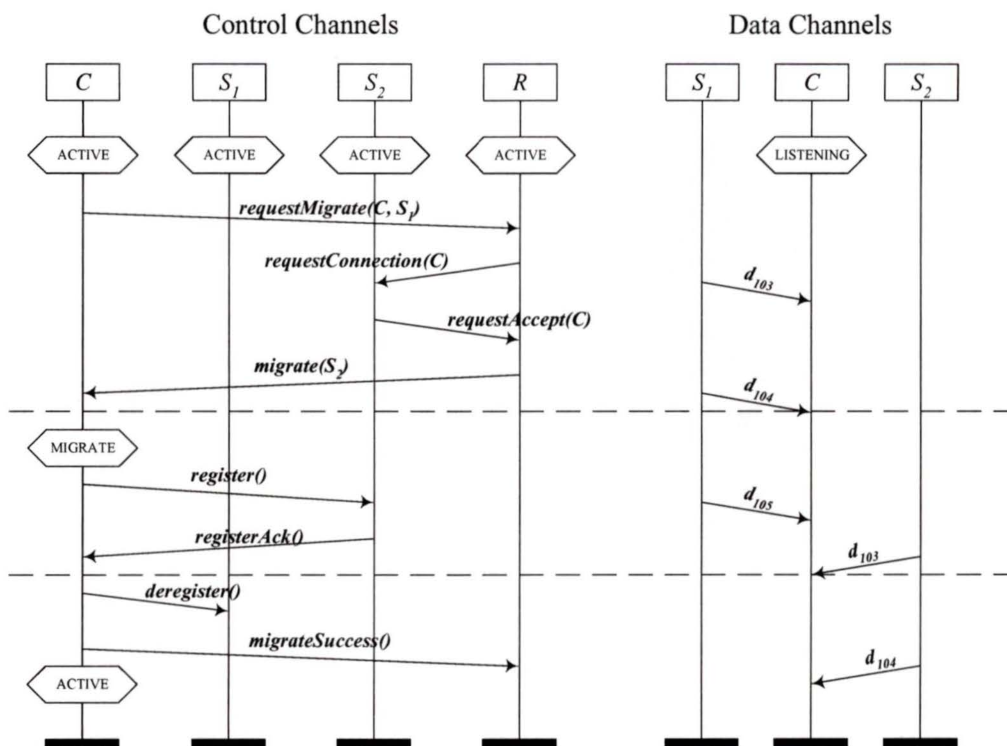


Figure 4.4: The message sequence chart showing the migration to a surrogate of two data units downstream from the existing surrogate

4.4.2 Invalid Surrogate Latencies

An invalid latency occurs at the start of migration when a client recognizes that the new surrogate's data stream is too far upstream or downstream from the existing surrogate's data stream to fit within the bounds of the data buffer. Experiments were performed where the client tried to migrate to a surrogate stream that did not fit within the bounds of its buffer.

Figure 4.6 shows the message sequence chart of a migration to a surrogate too far upstream. The surrogate S_2 had a data stream latency too far upstream from the existing surrogate so that it could not fit within the bounds of C's data buffer. C initially received continuous data from S_1. C initiated migration and the migration to S_2 proceeded. However, when C received the first data message from S_2 it discovered that S_2 had a stream latency that did not fit within the bounds of the buffer. Since the migration was invalid, C

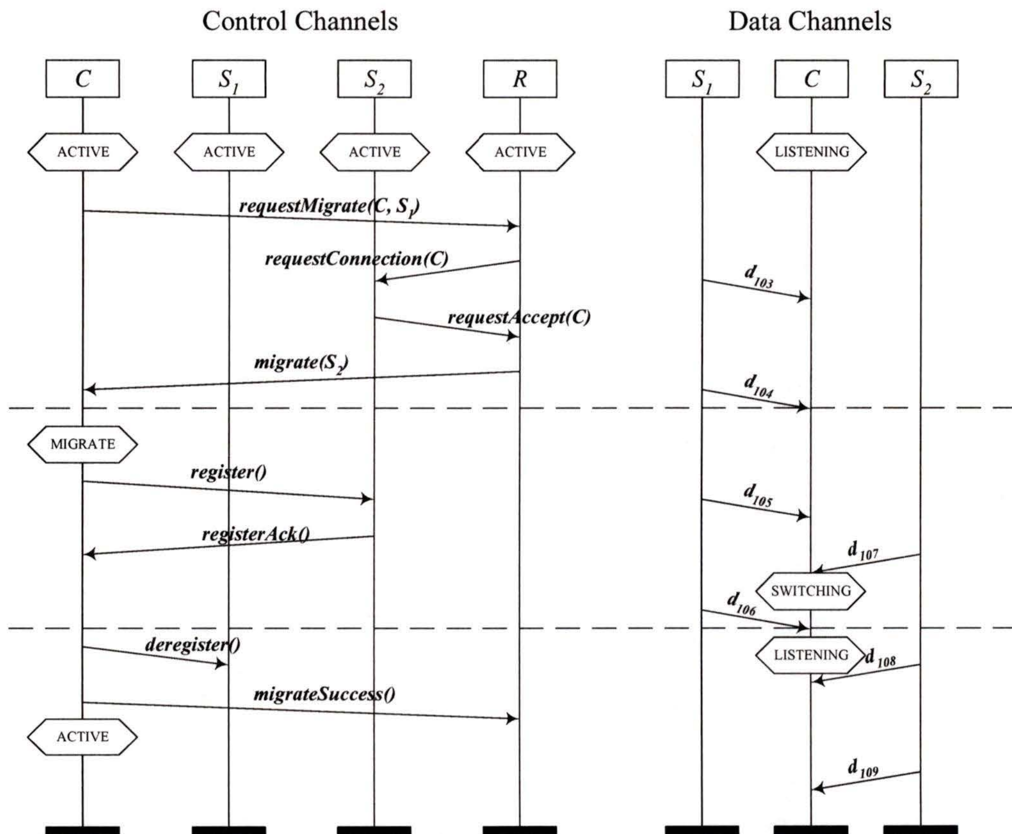


Figure 4.5: The message sequence chart showing the migration to a surrogate two data units upstream from the existing surrogate

sent the ***migrateInvalid()*** message to the coordinator R. The coordinator then requested the client to migrate to surrogate S_3. S_3's data stream fit within the bounds of C's buffer and the migration proceeded successfully.

4.4.3 Control Message Faults

Control message faults include lost control messages, duplicate surrogate migration requests, or logout from either the existing or new surrogates during migration.

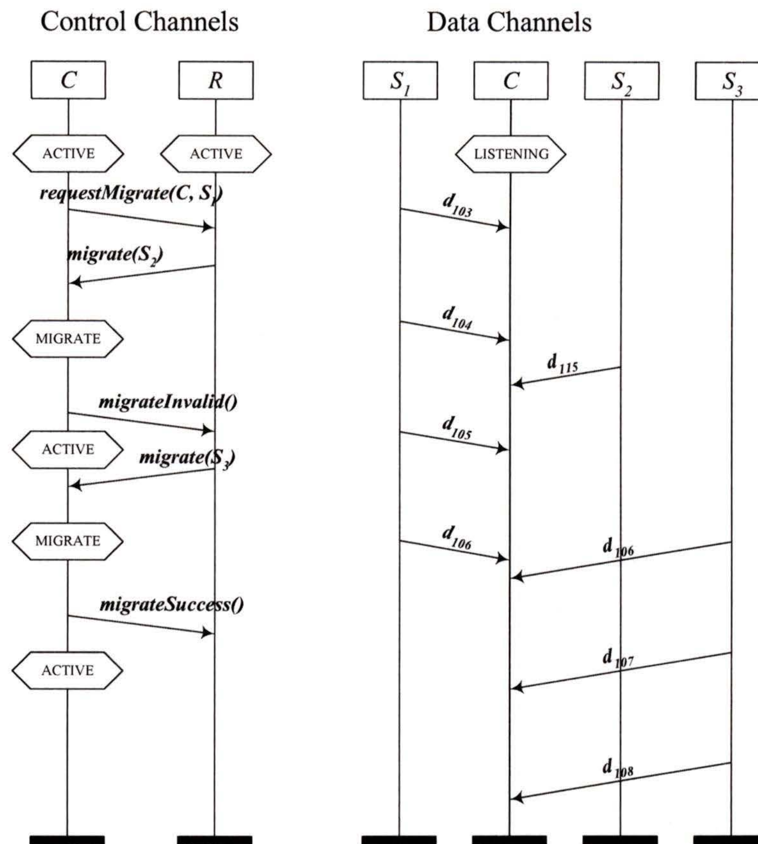


Figure 4.6: The message sequence chart showing client migration to a surrogate too far upstream (for simplicity control message communication to the surrogates has been removed)

4.4.3.1 Lost Control Messages

Lost control messages include the loss of any of the control messages shown in Figure 3.12. Timers are used to discover the lost control messages.

Seven experiments were created to emulate the loss of each of the seven data messages used in migration. The first experiment was the loss of the ***requestConnection(C)*** message from the coordinator R to the surrogate S_2. The timer on R timed out and the migration was cancelled.

The second experiment demonstrated the loss of the ***requestAccept(C)*** message from the surrogate S_2 to the coordinator R. This caused the timer on R to timeout and cancel

the migration. Since the migration did not continue, the timer on S_2 timed out and S_2 no longer waited for connection from the client C.

The third experiment was the loss of the ***migrate(S_2)*** message from R to C. Since C did not receive the message, no response was sent back to R. Eventually the timer on R timed out and R cancelled the migration. The timer on S_2 timed out and S_2 no longer waited for connection from C.

Figure 4.7 shows the message sequence chart for what happened when the ***register()*** message from C to S_2 was lost. The timer on S_2 timed out and stopped waiting for registration from C. The timer on C also timed out and it sent a ***migrationFail()*** message to R. After receiving this message R cancelled migration.

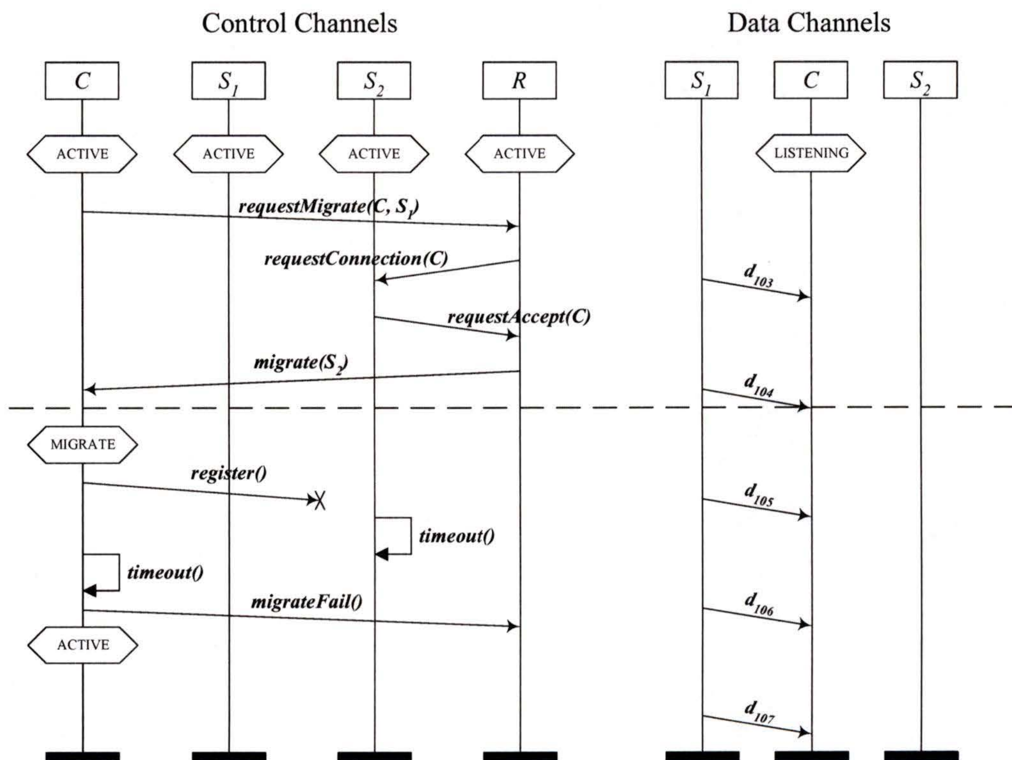


Figure 4.7: The message sequence chart showing a lost ***register()*** message from C to S_2

The fifth experiment demonstrated what happened when the ***registerAck()*** message

from S_2 to C was lost. Even though the message was lost the migration proceeded normally because the message was redundant. C knew that S_2 received registration as soon as it received data from S_2 on the data channel.

The sixth experiment demonstrated what happened when the ***deregister()*** message from C to S_1 was lost. This lost message did not affect the migration. However, the stream from S_1 to C was still alive. Even though the stream was still alive C ignored any data messages from S_1 and any ***heartbeat()*** messages. After not receiving replies to the ***heartbeat()*** message, S_1 assumed that C was not connected anymore and stopped sending C data.

The seventh experiment demonstrated what happened when the ***migrationSuccess()*** message from C to R was lost. Since the migration occurred successfully C was receiving from S_2. However, R timed out since it did not receive a response. It cancelled the migration and assumed it failed.

4.4.3.2 Duplicate Surrogate Migration Requests

As discussed previously, a duplicate surrogate error occurs if the coordinator requests the client to migrate to the same surrogate as its existing surrogate. A migration to a duplicate surrogate causes the client to respond to the coordinator with the message ***migrateDuplicate()***.

4.4.3.3 Logout from a Surrogate During Migration

During client migration logout from either the existing surrogate or the new surrogate may cause the migration to fail. Two experiments were performed so that the existing surrogate logged out during migration before the data buffer of the client had become continuous. The first experiment demonstrated the case where the data stream to the new surrogate had been successfully set up. When the logout came from the existing surrogate, migration was forced to complete and a ***migrateSuccess()*** message was sent to the coordinator. The forced migration introduced a gap between the junction of the existing stream and the new stream. This gap was removed but created a non-monotonic increasing order of sequence numbers

in the buffer. The second experiment demonstrated the case where the set up of the data stream to the new surrogate had failed. When a logout came from the existing surrogate, the client no longer had any streams, the migration failed and the client terminated.

An experiment was performed so that the new surrogate logged out during migration before the data buffer had become continuous. The logout caused a migration failure and a message ***migrateFail()*** was sent to the coordinator. However, the client was still able to continue to receive data since its stream connection to the existing surrogate had not yet been affected.

4.4.4 Data Stream Faults

The continuous data received by the client is important in progression of the client migration. A migration may fail if data stream faults occur with the existing stream or the new stream. Data stream faults include jitter, and lost data messages. If a data stream fault from the existing surrogate causes a migration timeout before the buffer is continuous then the migration will be forced to succeed if the new stream has been set up or will fail and cause client termination if the new stream has not been set up.

An experiment demonstrated messages failing to arrive from the existing surrogate after migration had started and the stream to the new surrogate had been set up successfully. The experiment ensured that the new stream was upstream from the existing surrogate. The migration timer timed out and migration was forced to succeed. The forced migration introduced a gap between the junction of the existing stream and the new stream. This gap was removed but created a non-monotonic increasing order of sequence number in the buffer.

A second experiment demonstrated messages failing to arrive from the existing surrogate after migration had started and the set up to the new surrogate's stream had failed. When the migration timer timed out there were no data streams available causing the migration to fail and the client to terminate.

Data stream faults for the new surrogate may not be detected during migration. To demonstrate the case for detection of data stream faults in the new stream an experiment

was created where no data messages arrived from the new surrogate after migration had started. Since no data messages arrived from S_2, no splice point was set, the migration timer timed out, C continued to receive data from S_1, and a ***migrateFail()*** message was sent to R.

4.5 Performance

Performance of the migration protocol is examined by looking at the number of messages and the time it takes for a migration to complete with various data frequencies and surrogate latencies. As well the light weight integration of the migration protocol is examined.

4.5.1 Performance Results

Seven or eight control channel messages are required for the migration protocol to function successfully. Seven control channel messages are required when the migration is initiated by the coordinator. Eight messages are required if the migration is initiated by a client or a surrogate; the extra message is for the surrogate or the client to request the migration to the coordinator.

The time in ms for completion of the migration protocol is dependent upon the speed, U, in ms of communication between network elements, the period, T, in ms between two data messages, and the latency difference, L, in message packets between a client's existing surrogate and the new surrogate. Due to when in the period a migration is initiated, the time, M_t, in ms varies on a range of size T. If $L \leq 1$ then the range is specified by the following equation.

$$4U + U + U < M_t < 4U + U + T + U$$
$$6U < M_t < 6U + T$$

Otherwise if $L > 1$ then the range is specified by the following equation.

$$4U + U + (L - 1) \times T + U < M_t < 4U + U + T + (L - 1) \times T + U$$
$$6U + (L - 1) \times T < M_t < 6U + L \times T$$

The $4U$ beginning each equation represents the time to send the initial request message to the coordinator by the client or surrogate, and the first three control channel messages of Figure 3.12. If the migration is initiated by the coordinator, $4U$ can be reduced to $3U$. The next U value for each of the above equations represents the time it takes for the client to send the registration message to the new surrogate. The last U value for each equation represents the time it takes the data message that causes buffer to become continuous. For the case where $L > 1$ then $(L - 1) \times T$ is the time it takes for the latency of the existing surrogate to reach the latency of the new surrogate to make the buffer continuous.

These formulas assume that $T >> U$, there is no computation time, and there is no jitter. Of course there will be computation time on the coordinator for the lookup of the element in a database, and computation time to splice both streams on the client side. In addition jitter could raise or lower the value of U.

Experiments were performed to measure the time of migration. All experiments had the five network elements set up as shown in Figure 4.1. Experiments were run stand alone and distributed. Stand alone experiments had the five network elements run on a single computer. Distributed experiments had each of the elements on a separate computer. Two of the computers ran Windows 2000, and the other three computers ran Linux. All computers were connected by a 10BT network. The two Windows 2000 computers took the job of the coordinator R and the origin O respectively. One Linux computer was dedicated to a role of the surrogate S_1. Depending on the experiment the other two Linux computers alternated as the client C and the surrogate S_2: these two computers were named $Crisp$ and $Arberon$. The computer $Crisp$ had a Pentium II 350Mhz processor and the computer $Arberon$ had a Pentium II 400 Mhz processor.

Table 4.1 shows the results of various experiments. Each row contains the summary of time results from three experiments; each of the three experiments contained anywhere from twenty-five to fifty migrations depending on the period T. The labels "SA" and "DI" stand for stand alone and distributed respectively. The results are collected on the node of the client.

The first column of the table describes the type of experiment performed. As dis-

		Initial Set Up(ms)	U (ms)	Min. $M_t(ms)$	Max. $M_t(ms)$
$L = 0$ $T = 20ms$	SA Crisp	0.079	N/A	10.002	27.253
	SA Arberon	0.071	N/A	10.007	20.800
	DI Crisp	1.606	0.382	2.437	22.283
	DI Arberon	1.350	0.320	2.494	22.294
$L = -5$ $T = 20ms$	SA Crisp	0.081	N/A	10.019	20.034
	SA Arberon	0.071	N/A	10.023	20.033
	DI Crisp	1.329	0.311	4.814	20.351
	DI Arberon	1.332	0.315	3.838	21.696
$L = 5$ $T = 20ms$	SA Crisp	0.081	N/A	90.006	100.024
	SA Arberon	0.071	N/A	88.873	100.024
	DI Crisp	1.811	0.433	84.854	100.305
	DI Arberon	1.326	0.314	83.790	101.808
$L = 0$ $T = 40ms$	SA Crisp	0.079	N/A	30.015	30.036
	SA Arberon	0.073	N/A	30.010	40.031
	DI Crisp	1.581	0.375	21.706	33.236
	DI Arberon	1.373	0.325	2.559	42.388
$L = -5$ $T = 40ms$	SA Crisp	0.079	N/A	20.020	40.021
	SA Arberon	0.071	N/A	40.007	40.021
	DI Crisp	1.340	0.316	30.754	36.027
	DI Arberon	1.337	0.316	10.858	39.752
$L = 5$ $T = 40ms$	SA Crisp	0.078	N/A	180.013	200.015
	SA Arberon	0.072	N/A	199.998	200.020
	DI Crisp	1.819	0.435	190.663	195.952
	DI Arberon	1.328	0.314	169.835	199.121

Table 4.1: Summary of migration timing results

cussed previously the variables L and T represent the latency difference in data messages between the existing surrogate and the new surrogate, and the period between data messages respectively.

The column "Initial Set Up (ms)" is the average time taken for the first four messages of migration. On the stand alone version this was strictly the cost of computation since messages in stand alone COOL are function calls. On the distributed version the time taken included the cost of computation, and the time taken to transport the UDP message. As can be seen from the stand alone experiments the machine $Arberon$ was slightly faster than $Crisp$.

The column U (ms) represents the average speed of sending one data message between two elements. This was determined for the distributed experiments by subtracting the cost of the initial set up of the stand alone experiment from the distributed experiment and then dividing the result by four. The value four is for each of the four initial migration messages. As can be seen from the experiments the cost to transport one message via UDP is just around 315 us. The results where the transport time of one message was larger could have been caused by jitter in the network, or from unpredictable process scheduling under Linux.

The final two columns represented the minimum and the maximum times of all the experiments to complete the migration; this was the value M_t in the above formulas. As can be seen from the results all the minimum and maximum times, except one, fit within the ranges of the above formula using L, T, and U. The one result that did not fit within the range is the maximum value of the stand alone $Crisp$ experiments. This probably was higher than the upper bound because of unpredictable process scheduling of Linux. As well the values of the last column for the distributed experiments may go outside the upper bound of the formula due to uncontrollable congestion in the network.

4.5.2 Light Weight

A protocol is light weight if there is no performance impact on the system when it is not used. The integration of the migration protocol into the content delivery system is relatively

light weight. This is determined by looking at the protocol's impact on each of the network elements.

The integration of the migration protocol did not affect the origin or the surrogate. The migration protocol used the same protocol to set up data streams as the client uses on initial stream set up. The use or non use of the migration protocol has no performance impact on the origin and the surrogates.

If the migration protocol is not used, there is no performance impact on the client. The migration routines require a ***migrate()*** message from the coordinator to get started and cause the client to perform execution of the protocol. If the client does not get this message then performance is not impacted. Once the client does get the migration message, most of the work is performed by the client to complete the migration. The client ensures there is progress during migration and that the two continuous data streams are correctly spliced together so that the strictly increasing sequence number order of messages is preserved.

If the migration protocol is not used, there is still a minor performance impact on the coordinator. When the coordinator validates a request response from a surrogate, it must perform a check to determine whether the request is an initial stream set up request or a migration request from the client. During migration the coordinator uses a timer to determine when the migration should complete. These two performance impacts on the coordinator are minimal.

Chapter 5

Concluding remarks

An adaptive client migration protocol has been designed and integrated into a continuous data network. When a client experiences degradation in its data stream to a surrogate, it then has the ability to migrate to another surrogate to improve its QoS. The adaptive client migration protocol adds a new level of resiliency to clients than previously existed with current continuous data networks.

The client migration protocol achieved all of the properties originally specified for it. It was demonstrated through various experiments that the client migration protocol had properties of distributed initiation, idempotency, and resiliency.

A generic continuous data delivery network was developed in the COOL language. The migration algorithm was integrated into this continuous data delivery network and experiments were shown to validate the migration protocol. The performance of the protocol was measured.

5.1 Future Work

The next logical step for this work is the implementation of the migration algorithm into a commercial continuous data delivery network. From a quick examination this algorithm has potential for practicality. For example a raw voice data stream requires a data rate of 64kbps [12]. During migration the client is receiving data streams from two servers so for

this example it must be able to handle a bandwidth of 128kbps. Most commercial cable and DSL networks are able to support this speed so this demonstrates that the algorithm is feasible.

A second area of investigation is to examine how to improve the chances of client migration to a surrogate of better QoS from the existing surrogate. Areas of improvement include the coordinator's measure of best, the ability of a client to choose from a list of alternative surrogates, and modification of the clients acceptance test.

The measure of best for the coordinator can be improved by periodically receiving resource and load updates from the surrogates. The surrogates could also send their sequence number position in the data stream. Then when a client requests for a migration the coordinator has more information to determine the best alternative surrogate for the client.

Currently the coordinator sends the ID of a single alternative surrogate in a migration request to the client. If the client has a problem with stream set up to the alternative surrogate, the client must request another migration. If instead the coordinator sends multiple alternative surrogates, this would speed up the client's search for a better surrogate.

A client's acceptance test could be narrowed from allowing the splice point to be set anywhere in the buffer to a smaller acceptance window. This smaller acceptance window would speed up the migration since it would take less time for the migration to become continuous. In addition, to reduce chances of buffer underrun and overrun, the acceptance window would not allow the buffer to go below or above specified capacities.

A third area for investigation is the use of this algorithm for fault tolerance. One example could be for a client to receive continuous data from two surrogates. When one data stream from a surrogate fails it could migrate to another surrogate to ensure that it would keep receiving from two surrogates. Adding the fault tolerance requirement to the coordinator requires some examination at the stream merging algorithm as it only accounts for handling jitter of a data stream but not failure.

References

[1] Sotirchos Stavros, Koziris Nectarios, and Papakostantinou George. A distributed media server management scheme. *Electrotechnical Conference, 2000. MELECON 2001. 10th Mediterranean*, 1:6–10, May 2000.

[2] Rob Glaser. Realsystem iQ transforming digital media delivery. Public Announcement Video Stream, http://www.realnetworks.com/realsystem/, December 2000. Date Viewed: April 5, 2001.

[3] Live broadcast distribution with Realsystem Server 8. RealSystem iQ Whitepaper, RealNetworks, http://www.realnetworks.com/realsystem/, December 2000.

[4] SHOUTcast online documentation. Nullsoft, http://www.SHOUTcast.com/support/docs/. Date Viewed: April 5, 2001.

[5] Gregory J. Conklin, Gary S. Greenbaum, Karl O. Lillevold, Alan F. Lippman, and Yuriy A. Reznik. Video coding for streaming media delivery on the internet. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3):269–281, March 2001.

[6] Randal C. Burns, Robert M. Rees, and Darrell D.E. Long. Efficient data distribution in a web server farm. *IEEE Internet Computing*, 5(4):56–65, July–August 2001.

[7] Junho Shim, Peter Scheuermann, and Radek Vingralek. Proxy cache design: Algorithms, implementation and performance. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):549–562, July–August 1999.

[8] Charles D. Cranor, Matthew Green, Chuck Kalmanek, David Shur, Sandeep Sibal, Jacobus E. Van der Merwe, and Cormac J. Sreenan. Enhanced streaming services in a content distribution network. *IEEE Internet Computing*, 5(4):66–75, July–August 2001.

[9] M. Day, B. Cain, G. Tomlinson, and P. Rzewski. A model for content internetworking. Technical report, Network Working Group Internet-Draft, February 2001.

[10] Dapeng Wu, Yiwei Thomas Hou, Wenwu Zhu, Ya-Qin Zhang, and Jon M. Peha. Streaming video over the internet: Approaches and directions. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3):282–300, March 2001.

[11] Eun Hwan Jo, Moon Hae Kim, and Jung-Guk Kim. Modeling of multimedia streaming services bassed on the tmo structuring scheme. *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 420–427, May 2001.

[12] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, third edition, 2000.

[13] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall International, Inc., 1996.

[14] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1998.

[15] Wanjiun Liao and Victor O. K. Li. The split and merge protocol for interactive video-on-demand. *IEEE MultiMedia*, 4(4):51–62, October–December 1997.

[16] Thinh PQ Nguyen and Avideh Zakhor. Distributed video streaming over internet. *Proceedings of SPIE Conference on Multimedia Computing and Networking*, January 2002.

[17] Andy Oram. Allcast: New life for live content. O'Reilly P2P, http://www.openp2p.com/pub/a/p2p/2001/07/17/allcast.html, July 2001. Date Viewed: February 11, 2002.

[18] Napster messages. http://opennap.sourceforge.net/napster.txt, August 2000.

[19] Stephen B. Wicker. *Error control systems for digital communication and storage*. Prentice-Hall International, Inc., 1995.

[20] Mantis H.M. Cheng, Gordon W. O'Connell, and Paul Wierenga. COOL : A Concurrent Object Coordination Language. Technical Report DCS-267-IR, Department of Computer Science, University of Victoria, Victoria, BC, Canada, July 2001.

[21] Jeff Schneider. Convergence of peer and web services. The O'Reilly Network, http://www.openp2p.com/pub/a/p2p/2001/07/20/convergence.html, July 2001.

[22] Realsystem Proxy 8 overview. RealSystem iQ Whitepaper, RealNetworks, http://www.realnetworks.com/realsystem/, December 2000.

[23] Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

Appendix A

The Migration Protocol

This appendix describes the migration protocol through the use of state-charts for the following three network elements: the surrogate, the coordinator and the client. The fourth network element, the origin, of the generic content distribution network used in this thesis is not involved in client migration. The messages accepted by the element at various states and the corresponding transitions are provided in a table. To aid in understanding the message and transition table, two description tables for the timers and variables, used by the element, are given.

A.1 Coordinator

The coordinator is responsible for setting up the network connections between clients, surrogates, and the origin. The state-chart shown in Figure A.1 and transition table shown in Table A.1 for the coordinator describes the messages used for migration. In finding the "best" surrogate for a client the coordinator may base its decision on one or more of the following properties: the geographic location of a surrogate with respect to a client, the network locale of a surrogate and a client, the number of hops from the client to a surrogate, the current network resources of a surrogate, the current computing resources of a surrogate, or the quality of the *data* channel between a surrogate and the client. Many of these properties require the coordinator to receive information updates from the surrogates.

The timers and variables used by the coordinator for migration are shown in Table A.2 and Table A.3 respectively.

For simplicity of the transition table repeated tries of migration for a client are shown with the ***migrateInvalid()*** message. Repeated tries may be extended to other failed migrations. If the coordinator performs repeated migrations for a client, it should store a history of the surrogates a client has visited. Without this a client could just repeatedly migrate back and forth between two surrogates.

If the coordinator has a reason to migrate a client it will send a ***requestMigrate(clients[i].id, surrogates[j].id)*** message to itself. If a migration is not already taking place for the client the coordinator will initiate a migration for that client.

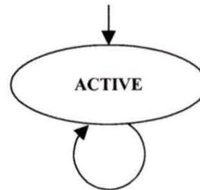


Figure A.1: The state-chart for the coordinator

Table A.1: Summary of coordinator messages and transitions

Current State	Next State	Message	Condition	Action(s)
¡initialize¿	ACTIVE			
ACTIVE	ACTIVE		*discover reason to migrate a client*	send ***requestMigrate(clients[i].id, surrogates[j].id)*** to **self**
ACTIVE	ACTIVE	***clients[i].mtimeout()***	from **clients[i].mtimer** and **clients[i].state**="MIGRATING"	**clients[i].state**:="RECEIVING", discard **clients[i].mtimer**
ACTIVE	ACTIVE	***clients[i].rtimeout()***	from **clients[i].rtimer** and **clients[i].state**="REQUESTING"	**clients[i].state**:="RECEIVING", **clients[i].rsurrogate**:=none, discard **clients[i].rtimer**
ACTIVE	ACTIVE	***migrateDuplicate()***	from **clients[i].id** and **clients[i].state**="MIGRATING"	discard **clients[i].mtimer**, **clients[i].state**:="RECEIVING"
ACTIVE	ACTIVE	***migrateFail()***	from **clients[i].id** and **clients[i].state**="MIGRATING"	discard **clients[i].mtimer**, **clients[i].state**:="RECEIVING"
ACTIVE	ACTIVE	***migrateInvalid()***	from **clients[i].id** and **clients[i].state**="MIGRATING"	find another "best" surrogate and store its ID into S_2, **clients[i].state**:="REQUESTING", send ***requestConnection(clients[i].id)*** to S_2, set **clients[i].rtimer**, **clients[i].rsurrogate**:=S_2
ACTIVE	ACTIVE	***migrateSuccess()***	from **clients[i].id** and **clients[i].state**="MIGRATING"	discard **clients[i].mtimer**, **clients[i].state**:="RECEIVING"
ACTIVE	ACTIVE	***requestAccept(client)***	from **surrogates[j].id** and **clients[i].id**=**client** and **clients[i].state**="REQUESTING" and **clients[i].rsurrogate**=**surrogates[j].id**	**clients[i].state**:="MIGRATING", discard **clients[i].rtimer**, **clients[i].rsurrogate**:=none, set **clients[i].mtimer**, send ***migrate(surrogates[j].id)*** to **clients[i].id**
ACTIVE	ACTIVE	***requestMigrate(client, S_1)***	from (**self**, **clients[i].id**, or **surrogates[j].id**) and **clients[i].id**=**client** and **surrogates[j].id**=S_1 and **clients[i].state**="RECEIVING"	**clients[i].state**:="REQUESTING", find alternative "best" surrogate not equal to S_1 and store its ID into S_2, send ***requestConnection(clients[i].id)*** to S_2, set **clients[i].rtimer**, **clients[i].rsurrogate**:=S_2
ACTIVE	ACTIVE	***requestReject(client)***	from **surrogates[j].id** and **clients[i].id**=client and **clients[i].state**="REQUESTING"	**clients[i].state**:="RECEIVING", **clients[i].rsurrogate**:=none, discard **clients[i].rtimer**

Timer	Timeout Message	Purpose
clients[i].mtimer	*clients[i].mtimeout()*	a timer to ensure a response returns from client migration, if no response comes back this timer times out and returns the client to its original state
clients[i].rtimer	*clients[i].rtimeout()*	a timer to ensure a request acknowledgement for client connection is returned from a surrogate

Table A.2: Summary of the timers used by the coordinator

Variable	Purpose
clients[]	a list of clients that can hold MAXCLIENT clients
clients[i].id	the ID of a client
clients[i].rsurrogate	the ID of the surrogate the coordinator sent a request for connection message
clients[i].state	holds the current state of the client, values include "RECEIVING", "REQUESTING", and "MIGRATING"
origin	holds the ID of the origin
S_2	temporarily holds the ID of the "best" surrogate
self	the ID of this coordinator
surrogates[]	a list of surrogates that can hold MAXSURROGATE surrogates
surrogates[i].id	the ID of a surrogate

Table A.3: Summary of the variables used by the coordinator

A.2 Surrogate

The surrogate relays continuous data to another surrogate or a client. The continuous data may be received from an origin or another surrogate. The state-chart describing messages used for migration by the surrogate can be found in Figure A.2. It is assumed that the surrogate has already initiated a connection to the origin or another surrogate. The messages and transitions for the state-chart are in Table A.4, the timers used by the surrogate are in Table A.5, and the variables used by the surrogate are in Table A.6.

The coordinator requests a client connection to the surrogate before the client may register to the surrogate. A **pendtimer** is available to timeout on failed registrations. Each client connection has a timer **hbtimer**, and a variable **hbcount**. This timer and variable provide a heartbeat function for the surrogate and ensure that a connection is still alive. When the counter **hbcount** is equal to 3 then the surrogate assumes that the receiver does not exist anymore and closes its connection to the receiver.

All messages are sent and received on the control message channels unless otherwise stated. For the surrogate the only messages sent and received on the data message channels are the ***data()*** messages.

If the surrogate has a reason to migrate a receiver it will send a ***requestMigrate(rcvr[i], self)*** message to the coordinator. If the receiver is a client and not already migrating the coordinator will initiate a migration for that client.

Figure A.2: The state-chart for the surrogate

Table A.4: Summary of surrogate messages and transitions

Current State	Next State	Message	Condition	Action(s)
¡initialize¿	ACTIVE			
ACTIVE	TERMINATE	***STOP()***		send ***deregister()*** to **sender**, send ***logout("surrogate")*** to **coord**, for all i where $0 \leq i < MAXRCVR$ if (**rcvr[i].state**="connected") send ***logout()*** to **rcvr[i].id**, for all i where $0 \leq i < MAXRCVR$ discard **rcvr[i]**
ACTIVE	ACTIVE		*discover reason to migrate a rcvr*	send ***requestMigrate(rcvr[i], self)*** to **coord**
ACTIVE	ACTIVE	***data(seqNo, payload)***	on data channel from **sender**	for all i where $0 \leq i < MAXRCVR$ **rcvr[i].state**="CONNECTED" send ***data(seqNo, payload)*** on data channel to **rcvr[i].id**
ACTIVE	ACTIVE	***deregister()***	from **rcvr[i].id**	discard **rcvr[i]**
ACTIVE	ACTIVE	***heartbeat()***	from **rcvr[i].id**	**rcvr[i].hbcount**:=0
ACTIVE	ACTIVE	***heartbeat()***	from **sender**	send ***heartbeat()*** to **sender**
ACTIVE	ACTIVE	***requestConnection(rcvrId)***	from **coord**, space available for another connection	**rcvr[i].id**:=**rcvrId** where i is location of empty space, **rcvr[i].state**:="PENDING", set **rcvr[i].pendtimer**, send ***requestAccept(rcvrId)*** to **coord**
ACTIVE	ACTIVE	***rcvr[i].pendtimeout()***	from **rcvr[i].pendtimer**	discard **rcvr[i]**
ACTIVE	ACTIVE	***rcvr[i].hbtimeout()***	from **rcvr[i].hbtimer**, **rcvr[i].hbcount**¡3	send ***heartbeat()*** to **rcvr[i].id**, **rcvr[i].hbcount**:=**rcvr[i].hbcount**+1
ACTIVE	ACTIVE	***rcvr[i].hbtimeout()***	from **rcvr[i].hbtimer**, **rcvr[i].hbcount**=3	send ***logout()*** to **rcvr[i].id**, discard **rcvr[i]**
ACTIVE	ACTIVE	***register()***	from **rcvr[i].id**	discard **rcvr[i].pendtimer**, send ***registerAck()*** to **rcvr[i].id**, set periodic **rcvr[i].hbtimer**, **rcvr[i].hbcount**:=0, **rcvr[i].state)**:="CONNECTED"
ACTIVE	ACTIVE	***requestConnection(rcvrId)***	from **coord**, no space available for another connection	send ***requestReject(rcvrId)*** to **coord**

Timer	Timeout Message	Purpose
rcvr[i].hbtimer	***rcvr[i].hbtimeout()***	a periodic heartbeat timer to see if the connection to **rcvr[i].id** is still alive
rcvr[i].pendtimer	***rcvr[i].pendtimeout()***	ensures there is a ***registration()*** message from **rcvr[i].id**

Table A.5: Summary of the timers used by the surrogate

Variable	Purpose
conectionList	holds a list of receivers that are currently receiving continuous data from the surrogate
coord	holds the ID of the coordinator
payload	usable data in a ***data(seqNo, payload)*** message
rcvrId	is the ID of a receiver
rcvr[]	represents a list of receivers of size MAXRCVR
rcvr[i].id	holds an ID of a receiver
rcvr[i].hbcount	a counter to keep count of lost heartbeat responses from the reciever
self	the ID of this surrogate
sender	the ID of the network element sending continuous data to the surrogate
seqNo	the sequence number of a ***data(seqNo, payload)*** message
rcvr[i].state	holds the state of the receiver, a "PENDING" state means that the surrogate is waiting for registration from the receiver, and a "CONNECTED" state means that the surrogate is delivering data to the receiver

Table A.6: Summary of the variables used by the surrogate

A.3 Client

The client receives continuous data from a surrogate. If it experiences jitter or congestion in the link to the surrogate the client will migrate to another surrogate. The state chart used for migration is shown in Figure A.3 and the transition messages used by the state chart are described in Table A.7. Table A.8 and table A.9 describe the timers and variables used by the client for migration. The functions for the buffer used by the client are described in Table A.10.

Initially the client has a connection to a surrogate. The coordinator initiates a migration to the client by sending the ***migrate(*S_2*)*** message. The client then makes a best effort to complete the migration. The migration succeeds when the existing stream and the new stream overlap, or the migration timer times out.

All messages are sent and received on the control message channels unless otherwise stated. For the client the only messages sent and received on the data message channels are the ***data()*** messages.

If the client has a reason to migrate from its existing surrogate it will send a ***request-Migrate(self,*** S_1***)*** message to the coordinator. If a migration has not already been initiated the coordinator will initiate a migration for the client.

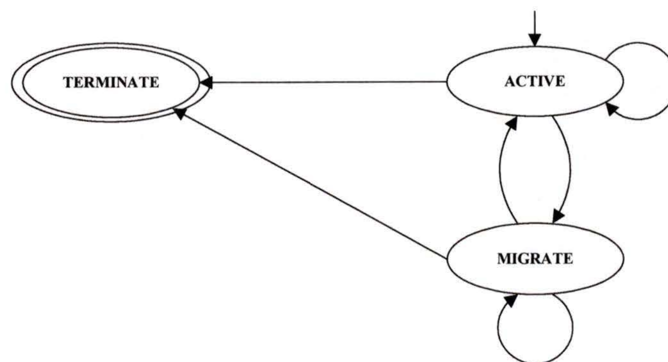


Figure A.3: The state-chart for the client

Table A.7: Summary of client messages and transitions

Current State	Next State	Message	Condition	Action(s)
¡initialize¿	ACTIVE			**dcState**="LISTENING"
ACTIVE	TERMINATE	***logout()***	from S_1	send ***logout("client")*** to **coord**
ACTIVE	TERMINATE	***STOP()***		send ***deregister()*** to S_1, send ***logout("client")*** to **coord**
ACTIVE	ACTIVE		*discover reason to migrate*	send ***requestMigrate(self,*** S_1***)*** to **coord**
ACTIVE	ACTIVE	***data(seqNo, payload)***	on data channel from S_1	**buf.put(seqNo, payload)**
ACTIVE	ACTIVE	***heartbeat()***	from S_1	send ***heartbeat()*** to S_1
ACTIVE	ACTIVE	***migrate(***S_2***)***	from **coord** and S_1=S_2	send ***migrateDuplicate()*** to **coord**
ACTIVE	MIGRATE	***migrate(***S_2***)***	from **coord** and S_1!=S_2	send ***register()*** to S_1, set **mtimer** to $freq \times MAXBUFSIZE$
MIGRATE	ACTIVE	***data(seqNo, payload)***	on data channel from S_1 and **dcState**="SWITCHING" and **buf.cont(seqNo,"EXISTING STREAM")**=true	**buf.put(seqNo, payload)**, discard **mtimer**, send ***deregister()*** to S_1, send ***migrateSuccess()*** to **coord**, **dcState**:="LISTENING", S_1:=S_2
MIGRATE	ACTIVE	***data(seqNo,payload)***	on data channel from S_2 and **dcState**="LISTENING" and **buf.inRange(seqNo)**=true and **buf.cont(seqNo, "NEW STREAM")**=true	**buf.putNew(seqNo, payload)**, discard **mtimer**, if (S_1!=none) send ***deregister()*** to S_1, send ***migrateSuccess()*** to **coord**, S_1:=S_2
MIGRATE	ACTIVE	***data(seqNo, payload)***	on data channel from S_2 and **dcState**="LISTENING" and **buf.inRange(seqNo)**=true and **buf.cont(seqNo, "NEW STREAM")**=false and S_1=none	**buf.putNew(seqNo, payload)**, **buf.mSuccess()**, discard **mtimer**, send ***migrateSuccess()*** to **coord**, S_1:=S_2
MIGRATE	ACTIVE	***data(seqNo,payload)***	on data channel from S_2 and **dcState**="LISTENING" and **buf.inRange(seqNo)**=false and S_1!=none	discard **mtimer**, send ***deregister()*** to S_2, send ***migrateInvalid()*** to **coord**
MIGRATE	ACTIVE	***logout()***	from S_1 and **dcState**="SWITCHING"	**buf.mSuccess()**, discard **mtimer**, send ***migrateSuccess()*** to **coord**, **dcState**:="LISTENING", S_1:=S_2

continued on next page

continued from previous page

Current State	**Next State**	**Message**	**Condition**	**Action(s)**
MIGRATE	ACTIVE	***logout()***	from S_2 and S_1!=none	**buf.mFail()**, **dcState**:="LISTENING", discard **mtimer**, send ***migrateFail()*** to **coord**
MIGRATE	ACTIVE	***mtimeout()***	from **mtimer** and **dcState**="LISTENING" and S_1!=none	**buf.mFail()**, discard **mtimer**, send ***deregister()*** to S_2, send ***migrateFail()*** to **coord**
MIGRATE	ACTIVE	***mtimeout()***	from **mtimer** and **dcState**="SWITCHING"	**buf.mSuccess()**, discard **mtimer**, send ***deregister()*** to S_1, send ***migrateSuccess()*** to **coord**, **dcState**:="LISTENING", S_1:=S_2
MIGRATE	TERMINATE	***data(seqNo,payload)***	on data channel from S_2 and **dcState**="LISTENING" and **buf.inRange(seqNo)**=false and S_1=none	discard **mtimer**, send ***deregister()*** to S_2, send ***migrateFail()*** to **coord**, send ***logout("client")*** to **coord**
MIGRATE	TERMINATE	***logout()***	from S_2 and S_1=none	discard **mtimer**, send ***migrateFail()*** to **coord**, send ***logout("client")*** to **coord**
MIGRATE	TERMINATE	***mtimeout()***	from **mtimer** and **dcState**="LISTENING" and S_1=none	discard **mtimer**, send ***deregister()*** to S_2, send ***migrateFail()*** to **coord**, send ***logout("client")*** to **coord**
MIGRATE	TERMINATE	***STOP()***		discard **mtimer**, if (S_1!=none) send ***deregister()*** to S_1, send ***deregister()*** to S_2, send ***migrateFail()*** to **coord**, send ***logout("client")*** to **coord**
MIGRATE	MIGRATE	***data(seqNo, payload)***	on data channel from S_1 and **dcState**="LISTENING"	**buf.put(seqNo, payload)**
MIGRATE	MIGRATE	***data(seqNo, payload)***	on data channel from S_1 and **dcState**="SWITCHING" and **buf.cont(seqNo, "EXISTING STREAM")**=false	**buf.put(seqNo, payload)**

continued on next page

continued from previous page

Current State	Next State	Message	Condition	Action(s)
MIGRATE	MIGRATE	***data(seqNo, payload)***	on data channel from S_2 and **dcState**="LISTENING" and **buf.inRange(seqNo)**=true and **buf.cont(seqNo, "NEW STREAM")**=false and S_1!=none	**buf.putNew(seqNo, payload)**, **dcState**:="SWITCHING", reset **mtimer** to $freq\times$**buf.s_minus_r()**
MIGRATE	MIGRATE	***data(seqNo, payload)***	on data channel from S_2 and **dcState**="SWITCHING"	**buf.putNew(seqNo, payload)**
MIGRATE	MIGRATE	***heartbeat()***	from S_1	send ***heartbeat()*** to S_1
MIGRATE	MIGRATE	***heartbeat()***	from S_2	send ***heartbeat()*** to S_2
MIGRATE	MIGRATE	***logout()***	from S_1	S_1:=none
MIGRATE	MIGRATE	***registerAck()***	from S_2	

Timer	Timeout Message	Purpose
mtimer	***mtimeout()***	a timer ensure that migration completes

Table A.8: Summary of the timers used by the client

Variable	Purpose
buf	stores the incoming data messages, the descriptions for the functions of this table are in the table below
coord	the ID of the coordinator
dcstate	the data channel state, it may hold the two values "LISTENING" or "SWITCHING", the data channel is only in the state "SWITCHING" when the data channel is receiving two data streams simultaneously
payload	usable data from a ***data(seqNo, payload)*** message
S_1	the ID of the existing surrogate
S_2	the ID of the new surrogate
self	the ID of this client
seqNo	the sequence number of a ***data(seqNo, payload)*** message

Table A.9: Summary of the variables used by the client

Function	Purpose
buf.cont(seqNo, "EXISTING STREAM")	return false if the sequence number does not cause the existing stream to overlap with the new stream, otherwise return true
buf.cont(seqNo, "NEW STREAM")	return true if this is the first sequence number of the new stream and it sets a splice point that is located between the read head and the write head
buf.inRange(seqNo)	return true if the **seqNo** fits within the range of the buffer, false otherwise, this function should only be used for the **seqNo** of the new surrogate
buf.mFail()	signal to the buffer that the migration will fail, this will cause any data from a new surrogate to be discarded
buf.mSuccess()	signal to the buffer that the migration is complete, the buffer will remove any gap between the new stream and the old stream, and the write head for the old stream will become the write head for the new stream
buf.put(seqNo, payload)	write the seqNo and the payload to the next position in the buffer, if the seqNo is less than the next expected sequence number discard the data, if there is no more space available discard the data. If there are two streams in the buffer and the data causes an overlap of the two streams, then set the write head to the write head prime.
buf.putNew(seqNo, payload)	If this is the first item from the new stream, set a splice point, write the data at the splice point and advance the write head prime one position forward of the splice point. If the splice point is in the past discard the remaining data forward of the splice point and the write head prime becomes the new write head. If this is not the first item from the new stream write the seqNo and the payload at the next position in the buffer referenced by the write head prime, if there is no space available then discard the data.
buf.s_minus_r()	return the distance between the splice point and the read head of the existing stream.

Table A.10: Summary of the client buffer functions

Vita

Surname: Howe Given Names: Anthony Justin

Place of Birth: Victoria, British Columbia, Canada

Educational Institutions Attended:

University of Victoria	1994-2001

Degrees Awarded:

B.Sc.	University of Victoria	1999

Honours and Awards:

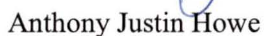
NSERC PGS A	1999-2001
ASI Scholarship	1999
University of Victoria President's Scholarship	1999-2001
TS McPherson Entrance Scholarship	1994-1999

Partial Copyright License

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Client Migration in a Continuous Data Network

Author: 

Anthony Justin Howe

February 11, 2002