On-Demand Multimedia Server Clustering Using Dynamic Content Replication

Jonathan Dukes

A thesis submitted for the degree of
Doctor of Philosophy in Computer Science

University of Dublin, Trinity College
Department of Computer Science

Supervisor: Dr Jeremy Jones

November 2004

Declaration

I declare that the work in this thesis has not been submitted for a degree at any other university, and that the work is entirely my own.

Signature

Jonathan Dukes

November 2004

Permission to Lend and/or Copy

I agree that the Library in Trinity College may lend or copy this thesis upon request.

Signature

Jonathan Dukes

November 2004

Acknowledgements

I would like to thank my supervisor, Dr Jeremy Jones, for his advice, patience, support and encouragement. I would also like to thank the other members of the Computer Architecture Group and the Department of Computer Science, particularly the technical and support staff, for their help.

The work described here has been part-funded by Trinity College Dublin and Microsoft Research in Cambridge and I am grateful for their support.

I want to thank my friends, especially John, Tara and Andrew, for providing a welcome distraction during the final stages of the preparation of this thesis. Finally, I would not have been able to finish this thesis without the support and encouragement of my parents Val and Neville, my sister Karen and my uncle Clive.

Abstract

This thesis examines the provision of on-demand multimedia streaming services using clusters of commodity PCs. In the proposed *HammerHead* multimedia server cluster architecture, a dynamic content replication policy is used to assign non-disjoint subsets of the presentations in a multimedia archive to cluster nodes. Replicas of selected presentations can be created on more than one node to achieve load-balancing, increase performance or increase service availability, while avoiding complete replication of a multimedia archive on every node. Since the relative demand for the presentations in a multimedia archive will change over time, the assignment of presentations to nodes must be periodically reevaluated.

A group communication system is used by the HammerHead architecture to implement a cluster-aware layer, which maintains the aggregated state of the commodity stand-alone multimedia server on each cluster node. The aggregated cluster state is used to redirect client requests to specific nodes and to implement the dynamic content replication policy. By replicating the aggregated cluster state on each node, the client redirection task can be shared and the implementation of the dynamic replication policy can tolerate multiple node failures. The HammerHead architecture proposed in this thesis is the first multimedia server cluster architecture to combine the use of group communication with the implementation of a dynamic replication policy.

The *Dynamic RePacking* content replication policy, which has been used in the prototype HammerHead server cluster, is a significant improvement of the existing MMPacking replication policy. Dynamic RePacking separates replication to achieve

load-balancing from replication to increase the availability of selected multimedia presentations, allowing increased service availability and performance to be traded against storage cost. In addition, replicas are assigned to nodes in a manner that allows load-balancing to be maintained when nodes fail.

Performance results obtained from a prototype HammerHead cluster and from an event-driven simulation show that Dynamic RePacking achieves a level of performance close to that achieved by replicating an entire multimedia archive on every cluster node, while significantly reducing the required storage capacity. It is believed that this is the first study to evaluate the performance and behaviour of a dynamic replication policy, outside a simulation environment.

Contents

Declaration ii

Permission to Lend and/or Copy iii

Acknowledgements iv

Abstract v

1 Introduction 1

1.1 On-Demand Multimedia Server Clustering 1

1.2 Thesis Contributions 3

1.3 Overview 5

2 On-Demand Multimedia Streaming Services 6

2.1 Multimedia Streaming 7

2.1.1 Characteristics 7

2.1.2 Network Protocols 9

2.2 On-Demand Multimedia Streaming Services 13

2.2.1 Client Behaviour 16

2.3 On-Demand Multimedia Server Design Considerations 20

2.3.1 Scalability 21

2.3.2 Availability 22

2.3.3 Storage Cost 26

2.4 On-Demand Multimedia Server Architectures 27

2.4.1 Single-server Model 27
2.4.2 Cloned Server Model 28
2.4.3 Parallel Multimedia Streaming Servers 31
2.4.4 Dynamic Replication 38
2.4.5 Storage Area Networks 41
2.4.6 Content Distribution Networks and Caching Proxy Servers . . . 42
2.4.7 Multicast Stream Delivery 43
2.5 Summary 44

3 Dynamic Replication and Group Communication in Multimedia Server Clusters 46

3.1 Dynamic Replication Policies 47
3.2 Group Communication in Multimedia Server Clusters 57
3.3 Summary 60

4 Dynamic RePacking 63

4.1 Demand Estimation 64
4.2 Replica Assignment 67
4.2.1 MMPacking 67
4.2.2 Dynamic RePacking 71
4.2.3 Constrained Storage Capacity 78
4.2.4 Fault-Tolerant Dynamic RePacking 80
4.2.5 Fit Threshold 84
4.3 Simulation 85
4.4 Summary 85

5 The HammerHead Server Cluster 86

5.1 Group Communication using Ensemble and Maestro 87
5.1.1 View-Synchronous Group Communication 89
5.1.2 The Ensemble and Maestro Toolkits 92

5.2 Windows Media Services 94
5.3 Network Load-Balancing Cluster Service 96
5.4 HammerHead Architecture 96
5.4.1 `HammerSource` Component 97
5.4.2 `HammerServer` Component 104
5.4.3 Group Communication in HammerHead 114
5.5 Summary 117

6 HammerHead Performance 119

6.1 Performance Metrics 120
6.2 HammerHead Prototype Performance Evaluation 122
6.2.1 Prototype Experiment 1: Replication Policy 127
6.2.2 Prototype Experiment 2: Target Cluster Load 134
6.2.3 Prototype Experiment 3: Changing Presentation Popularity . . . 137
6.2.4 Prototype Experiment 4: Reduced Storage Capacity 140
6.2.5 Prototype Experiment 5: Heterogeneous Presentations 143
6.2.6 Prototype Experiment 6: Availability 146
6.3 Dynamic RePacking Simulation 152
6.3.1 Simulation Experiment 1: Comparison of Simulation and Prototype Results 153
6.3.2 Simulation Experiment 2: Increasing Archive Size 159
6.3.3 Simulation Experiment 3: Increasing Cluster Size 161
6.3.4 Simulation Experiment 4: Popularity "Skew" Parameter θ 164
6.3.5 Simulation Experiment 5: Heterogeneous Cluster Configurations 166
6.4 Summary 170

7 Conclusions and Future Work 173

7.1 Future Work 175
7.2 Final Remarks 177

A Calculating Mean-Time-To-Service-Loss 179

Bibliography 183

List of Tables

4.1 Meaning of `packMap` values before Dynamic RePacking 74

4.2 Meaning of `packMap` values after Dynamic RePacking 74

5.1 Windows Media Services events captured by the HammerHead event notification plug-in 95

5.2 HammerHead `Stream` categories 100

5.3 HammerHead events issued by `HammerSource` plug-ins to the `HammerServer` cluster-aware layer 101

6.1 Parameters used for the replication policy experiment 128

6.2 Parameters used for the target cluster load experiment 136

6.3 Parameters used for the changing popularity experiment 139

6.4 Parameters used for the reduced storage capacity experiment 141

6.5 Parameters used for the heterogeneous presentations experiment 144

6.6 Parameters used for the availability experiment 148

6.7 Parameters used to evaluate the effect of increasing the archive aize . . 159

6.8 Parameters used to evaluate the effect of increasing the cluster size . . . 163

6.9 Parameters used to evaluate the effect of changing the popularity skew parameter, θ 166

6.10 Parameters used in heterogeneous cluster simulation experiments 167

6.11 Heterogeneous cluster configurations A and B, and equivalent homogeneous configuration X 168

6.12 Heterogeneous cluster configurations C, D and E, and equivalent homogeneous configuration Y 168
6.13 Results of the heterogeneous cluster experiments 169

List of Figures

2.1 The TCP/IP reference model [Tan96] 9

2.2 RTSP request 11

2.3 RTSP response 12

2.4 Illustration of an RTSP session 14

2.5 Zipf Distribution for $K = 100$ and $\theta = 0$, $\theta = 0.5$ and $\theta = 1$. 17

2.6 Server resource utilization for presentations with different durations and bit-rates. 19

2.7 Simple four node web server cluster 21

2.8 Replication–partition data distribution spectrum 26

2.9 Single server model 28

2.10 Cloned server model 29

2.11 Parallel server model 31

2.12 Parallel server proxy configurations 33

2.13 Tiger disk layout 35

2.14 Failure of Tiger nodes 36

2.15 Dynamic replication server model 39

2.16 Storage Area Network (SAN) architecture for on-demand multimedia streaming 41

2.17 Content distribution network 42

4.1 Saved measurements of demand used to estimate $D'_{i,t}$ 66

4.2 Illustration of the MMPacking algorithm, showing the popularity assigned to presentations with more than one replica and the popularity of each node 69
4.3 MMPacking algorithm 69
4.4 List of `Presentation` objects used by Dynamic RePacking 72
4.5 List of `Node` objects used by Dynamic RePacking 72
4.6 Illustration of the use of `Node` and `Presentation` objects 75
4.7 Pseudo-code for the `PackReplica()` function 77
4.8 Pseudo-code for the Dynamic RePacking algorithm 78
4.9 Illustration of the Dynamic RePacking algorithm showing the target shortfall of each node 79
4.10 Dynamic RePacking over two phases for a four-node cluster 82

5.1 View-synchronous group communication [CDK01, Bir96] 90
5.2 Conceptual illustration of state transfer 91
5.3 Maestro pull-style state transfer [Vay98] 94
5.4 HammerHead architecture 97
5.5 Illustration of the HammerHead state information for the Windows Media Services instance on a single cluster node 99
5.6 HammerHead `Replica` state transitions 103
5.7 HammerHead client redirection 107
5.8 Measurement of stream duration 109
5.9 HammerHead replica creation 110
5.10 Template `Replica` and new `Replica` objects containing "cookie" information 111

6.1 Master-slave architecture of workload generator 123
6.2 HammerHead performance evaluation environment 126
6.3 Results of the replication policy experiment 130
6.4 Results of the target cluster load experiment 135

6.5 Results of the changing popularity experiment 138
6.6 Results of the constrained storage capacity experiment 142
6.7 Results of the heterogeneous presentations experiment 145
6.8 Achieved service load after node failure 147
6.9 Harvest after node failure 148
6.10 Fault-tolerant and non-fault-tolerant Dynamic RePacking 151
6.11 Comparison of simulation and prototype results for the replication policy experiment 154
6.12 Comparison of simulation and prototype results for the target cluster load experiment 156
6.13 Comparison of simulation and prototype results for the changing presentation popularity experiment 157
6.14 Comparison of simulation and prototype results for a cluster with reduced storage capacity 158
6.15 Results of the archive size experiment 160
6.16 Results of the cluster size experiment 162
6.17 Results of the popularity skew experiment 165

Chapter 1

Introduction

1.1 On-Demand Multimedia Server Clustering

The prevailing model for the provision of scalable, highly-available services in many applications domains is the server cluster [FGC+97]. The Google web search engine [BDH03] is a good example of the use of this model. The service is provided by clusters of over 15,000 commodity PC nodes. Incoming client requests are distributed among web servers – which manage the execution of queries – and the execution of each query is further distributed among many cluster nodes, providing fast response times. The index data required by the search engine is partitioned to facilitate the parallel execution of requests and replicated to make the service resilient to node failures. The availability and scalability of the service can be increased efficiently, without interruption to service, through the addition of new nodes and further replication of the index data.

The server cluster model may also be adopted to implement an on-demand multimedia streaming service. In an on-demand streaming environment, remote users can select a multimedia presentation, such as an audio or video clip, from an archive and have the presentation rendered locally by a multimedia player. The data stream containing the presentation is transported across a network at a rate that is at least sufficient to render the multimedia presentation at the desired playback rate, as the

data arrives. True on-demand multimedia streaming gives users interactive control of the playback of presentations, with the ability to pause, rewind, fast-forward or move directly to a point of interest in a presentation. The provision of a service such as this typically requires a separate stream for each user session, making the provision of true on-demand multimedia streaming expensive in terms of network and server resources.

Commodity software solutions for providing on-demand streaming services, such as RealNetworks' Helix Universal Server [Rea03] or Microsoft's Windows Media Services [GB03], are easily deployed. Frequently, however, the demand for an on-demand multimedia streaming service will exceed the capacity of a single computer, particularly when commodity PCs are used. Providing additional service capacity requires the aggregation of the resources of multiple PCs, forming a server cluster.

The distribution of multimedia content among the nodes in a cluster is of critical importance. One approach that may be adopted is to *clone* both the multimedia server and the entire archive on each cluster node. When a client request arrives, the request is serviced by the least loaded node in the cluster, evenly distributing the workload among the nodes to maximize the number of streams that can be supplied. Cloning large multimedia archives – which may be many terabytes in size – in this manner on large numbers of cluster nodes becomes prohibitively expensive as the size of the archive increases. While the cost-per-gigabyte of disk storage equipment continues to decrease, the manpower required to manage such large archives, replicated many times, becomes an important consideration [GS00], as do environmental issues such as cooling and power consumption [BDH03], and the time taken to replicate the entire archive when adding new nodes or replacing failed ones.

The challenge, therefore, is to reduce the storage capacity required on each cluster node by avoiding complete replication of all content, while still allowing the client workload to be distributed evenly across the nodes. Wide data striping or server striping [Lee98] – which is conceptually similar to RAID 0 [PGK88] and has been extensively investigated in the past – divides each multimedia stream into blocks and distributes

these blocks across the nodes in a cluster. This approach, however, has several limitations. In particular, it suffers from poor availability, unless redundant data is stored, and limited scalability [CGL00].

This thesis examines in detail the use of an alternative technique for distributing a multimedia archive among cluster nodes and addresses both the performance of the technique and, in particular, its implementation in a server cluster environment. Rather than cloning an entire multimedia archive on every cluster node, a subset of the presentations in the archive is assigned to each node, with selected presentations replicated on more than one node to facilitate load balancing or to increase availability. This approach, which has been referred to in the past as *dynamic replication* [LLG98], requires that the distribution of content be either continuously or periodically reevaluated to adapt to changing client request patterns, facilitate load balancing and adapt to changes in the configuration of the cluster, such as the addition or removal of nodes. It will be argued that dynamic replication has the potential to offer the best compromise between scalability, availability, performance and cost in an on-demand streaming service provided by a loosely-coupled cluster of commodity PCs. In particular, dynamic replication can take advantage of the falling cost-per-gigabyte of disk storage equipment, without the disadvantages of replicating the entire archive on every node, allowing increased availability and performance to be traded against storage cost.

1.2 Thesis Contributions

Dynamic replication has been studied in the past and a number of dynamic replication policies have been proposed. This thesis examines dynamic replication in the context of more recent experiences with the provision of large-scale internet services and argues that dynamic replication is the most suitable content distribution technique for implementing large-scale on-demand multimedia streaming services, using the server cluster model.

An existing dynamic replication policy has been adopted and significantly improved to reduce the cost of adapting to changes in client behaviour or cluster configuration. An important feature of the proposed policy, called *Dynamic RePacking*, is the separation of replication to achieve load-balancing – based on the resources required by the expected number of concurrent streams of each presentation – from replication to increase the availability of selected multimedia presentations – based, for example, on the relative importance of each presentation. Separating these goals increases the flexibility of the replication policy and allows allows a configurable trade-off between partitioning and replication of multimedia content to be achieved. A novel feature of Dynamic RePacking is the assignment of replicas to cluster nodes in a manner that allows load-balancing to be maintained when nodes fail or are otherwise removed from the cluster.

Although other dynamic replication policies have been proposed in the past, no account of an implementation of any of these policies could be found in the literature. This thesis describes the architecture of the HammerHead on-demand multimedia streaming server cluster, which uses the proposed Dynamic RePacking policy to periodically determine a suitable assignment of presentations to cluster nodes. The HammerHead architecture has a number of novel features. It implements a cluster-aware layer on top of existing commodity multimedia server software, taking advantage of the features of commodity solutions and their ability to interact with unmodified commodity multimedia players, while allowing them to scale beyond the service capacity of a single node. The cluster-aware layer makes use of a group communication service [ACM96] to maintain the aggregated state of each commodity multimedia server instance in the cluster. In the HammerHead prototype, this aggregated state is used to implement the Dynamic RePacking content replication policy and to provide a client redirection service, redirecting initial client requests to suitable cluster nodes. The redirection of clients takes account of the location and state of the replicas of each presentation in the multimedia archive and the current activity of each cluster node. The aggregated state is replicated on cluster nodes, allowing the client redirection role to be shared among

the cluster nodes and allowing the implementation of the Dynamic RePacking policy to continue in the presence of multiple node failures. It is believed that HammerHead is the first multimedia server cluster architecture to combine the use of group communication with the implementation of a dynamic replication policy. It is expected that the HammerHead architecture will provide a framework for the future implementation of existing multimedia server policies, such as admission control and multicast batching, on a cluster-wide basis.

The performance of the HammerHead architecture and the Dynamic RePacking content replication policy has been extensively evaluated through experimentation with a prototype HammerHead cluster. It is believed that this is the first study to analyze the performance and behaviour of dynamic replication outside a simulation environment. An event-driven simulation has also been used to evaluate the performance of the Dynamic RePacking policy for a wider range of cluster configurations and client workloads than was possible using the prototype cluster.

1.3 Overview

Chapter 2 describes the problem area in more detail, addressing issues including the provision of on-demand multimedia streams over a network, the behaviour of the clients accessing the service and the requirements of the service. Multimedia streaming server architectures based on four different content distribution techniques are considered and an argument for the use of selective dynamic replication is presented. Chapter 3 describes existing dynamic replication policies and past uses of group communication in the implementation of on-demand multimedia streaming services. In Chapter 4, the Dynamic RePacking policy proposed by this thesis is described in detail. The design and implementation of the HammerHead multimedia server cluster is described in Chapter 5. An evaluation of the performance of the prototype HammerHead cluster and the Dynamic RePacking policy is presented in Chapter 6. Finally, conclusions and areas for future work are discussed in Chapter 7.

Chapter 2

On-Demand Multimedia Streaming Services

This chapter describes the nature of multimedia streaming and the resulting implications for the design of on-demand multimedia streaming services.

First, the characteristics of a single media stream are described. Transmitting such a stream over a network for playback by a client requires the use of network protocols to allow clients to control the stream and to transport the multimedia content from the server. The RTSP control protocol and the RTP transport protocol are presented as examples.

An on-demand multimedia streaming service makes an archive of multimedia presentations[1] available for streaming to clients over a network. Before designing servers to provide a service such as this, it is important to consider the typical behaviour of the clients that will use it. In particular, the relative popularity of individual presentations and the effect of the bit-rate and average duration of streams of a presentation are considered.

[1]The term "presentation" is used throughout this thesis to refer to a multimedia presentation, such as a movie or audio track, that can be delivered to a client by an on-demand multimedia streaming service. Several instances or copies of a presentation may be stored in a server cluster and such instances are referred to as "replicas". Streams of a presentation, of which there are replicas on two nodes in a cluster, can be provided from either node.

Finally, four architectural models, which correspond to different content distribution techniques, can be used to provide an on-demand multimedia streaming service. These models are described and the advantages and disadvantages of each model are discussed in terms of performance, availability, scalability and cost. It will be argued that selective dynamic replication is the most appropriate data distribution technique when constructing a loosely-coupled server cluster to provide an on-demand streaming service. Existing dynamic content replication policies will be described in more detail in Chapter 3.

2.1 Multimedia Streaming

Multimedia streaming is the delivery of content, for example a presentation containing audio and video, across a computer network, for playback by a multimedia player application on a client computer. Before attempting to address the design and implementation of a service to supply multimedia streams to clients *on-demand*, it is necessary to understand the characteristics of a single multimedia stream and the requirements for the transmission of such a stream across a network.

2.1.1 Characteristics

A rendered media stream may appear, from a user's perspective, to be a continuous audio or video clip. A multimedia stream can contain a combination of audio, video, images, text or other media types, depending on the nature of the presentation. In order to store continuous multimedia content on a digital medium (for example, disk, tape or RAM), however, the content must be encoded in a discrete format.

In the case of a video clip, an ordered sequence of static images or *frames* is rendered in succession to give someone viewing the stream the impression of a changing scene. The number of frames rendered per second is referred to as the *frame rate*. A typical frame rate for a TV quality video stream is 25 frames per second. Each frame of a video stream contains a fixed number of pixels, determined by the *resolution*

(horizontal and vertical dimensions) of the frame. If the resolution of a video stream is 640×480 pixels and each pixel is encoded using 24-bit values to represent different colours, the data size of each frame is just under 0.9MB. If 25 frames are displayed each second, just over 184 megabits of data needs to be delivered to the player every second, either from a local storage device or over a network. This value is the *data-rate* or *bit-rate* of the media stream.

Similarly, the amplitude of a sound wave can be sampled at a fixed rate and the amplitude stored as one of 2^n distinct values, where n is the number of bits used to store each sample. For example, telephone quality sound is represented by 8-bit values sampled at 8kHz or 8,000 times per second. Stereo CD quality sound is represented by 16-bit values sampled at 44kHz for each of two channels. The corresponding data rates are 64kbps and 1408kbps respectively.

The data rate of a multimedia stream and its duration (the time taken to play back the stream) will together determine the storage capacity required to store the presentation. Storing and transmitting large quantities of uncompressed multimedia data is often impractical, requiring the data to be stored and transmitted in a compressed form. The MPEG standards are examples of multimedia compression standards that provide for the compression of both audio and video media types. MPEG and other compression standards are discussed by Buford [Buf94].

Regardless of the media type, the frames or samples need to be available to the player in time for the video, sound or other media type to be decompressed and rendered. If a frame or sample arrives too late, the viewer will experience a frozen image in the case of a video stream or a silence in the case of an audio stream. Because the data in a multimedia stream needs to be made available to players according to a schedule determined by the data rate of the stream, multimedia streaming is subject to real-time constraints.

Most multimedia players use buffering techniques to reduce the effect of delayed frames or samples on perceived performance. When a stream is started, the player will accumulate a relatively small number of packets before starting to render the stream.

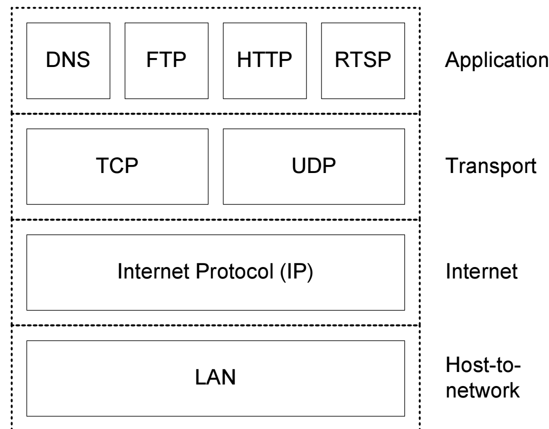


Figure 2.1: The TCP/IP reference model [Tan96]

Thus, packets may be delayed by up to the length of the buffer and still arrive in time to be rendered according to the stream's playback schedule. Any delay beyond the length of the buffer will result in a loss of quality rather than the failure of the stream. For these reasons, multimedia streaming is considered to be a *soft real-time* application.

2.1.2 Network Protocols

Network protocols facilitate the transfer of data across computer networks. A typical application will communicate using several protocols, arranged as layers in a stack. The TCP/IP reference model [Tan96], illustrated in Figure 2.1, describes an arrangement of protocol layers used for communication over local area networks and the Internet.

The model describes the protocols required to implement a *packet-switching* network. A message to be transmitted across a network is placed in one or more packets, depending on the size of the message. Each packet contains the destination address and is transmitted independently from the source to the destination. Packets may arrive in a different order to that in which they were sent, or they may not arrive at all. This functionality is provided in the internet layer and the protocol used is the Internet Protocol or IP.

While the internet layer allows hosts on the network to send and receive packets, the transport layer allows related applications on those hosts to communicate with each

other by sending messages. Two protocols are used in the transport layer. The first is the Transmission Control Protocol or TCP, which allows applications to establish a reliable connection over which messages can be sent and received in order, using the packet transmission facility provided by the internet layer. Some applications do not require the reliable, ordered communication of messages provided by TCP so a second User Datagram Protocol (UDP) is also provided. Both of these protocols define ports to allow messages intended for a specific application on a host to be delivered to that application.

The top application layer in the TCP/IP model contains those protocols defined by applications. Examples of application protocols are the Hypertext Transfer Protocol (HTTP) used to transmit messages on the World Wide-Web and the Domain Name Service (DNS) protocol used to communicate with servers that map host names to network addresses.

A multimedia streaming application typically requires at least two application layer protocols. The first category of protocol is required to transport stream data across a network. For most streaming applications, it is more important that the packets of data containing the stream arrive on time, according to the playback schedule of the server, rather than guaranteeing that all of the packets are delivered reliably. In such circumstances the resources required by TCP to implement reliable, connection-oriented communication are largely wasted and will reduce the overall capacity of a multimedia streaming system. UDP is a more suitable transport protocol upon which an application level protocol for transporting multimedia content can be built.

UDP does not implement reliable message delivery, so the protocol incurs a lower overhead than TCP and the loss of occasional packets will not usually cause the stream to fail and will instead result only in a temporary loss of quality. Since UDP does not provide message ordering – without which there is no way for a player application to determine the order in which packets should be processed for rendering – an application layer protocol is required to provide this information. The Real-time Transport Protocol (RTP) [SRL98b] is one such protocol. Each RTP packet has a header containing a

```
DESCRIBE rtsp://myserver/movies/movie1.wmv RTSP/1.0
CSeq: 1
```

Figure 2.2: RTSP request

sequence number, allowing packets to be processed and rendered in the correct order. The header also identifies the media type (e.g. audio or video) contained in the packet. RTP sends the different media types contained in a multimedia presentation as separate streams, with each stream identified by a different payload type. To allow the playback of the different streams to be synchronised, the RTP header also contains a time stamp that corresponds to the playback time of the first byte of data contained in the packet.

While protocols such as RTP facilitate the transport of multimedia streams across a network, a second category of protocol is required to establish, control and close the stream. Stream control protocols are of more interest, in the context of this thesis, than stream transport protocols and this will become apparent in later chapters. The Real Time Streaming Protocol (RTSP) [SRL98a] is one such protocol, used for the control of both on-demand and interactive streams. RTSP will be referred to in later chapters and the relevant parts of the protocol are described here.

RTSP is similar to the HTTP/1.1 [FGM+99] protocol used by the World-Wide Web, both in terms of syntax and functionality. It is a text-based protocol and an RTSP session takes the form of request-response exchanges between a client and a server. Unlike HTTP, RTSP clients and servers can both send requests. Each RTSP request contains a method; an identifier for the multimedia presentation or stream to which the method applies, in the form of a Uniform Resource Identifier (URI); the RTSP version number; a request sequence number to identify the request within a session; and zero or more additional parameters. For example, Figure 2.2 illustrates an RTSP request sent by a client to a server to obtain a description of a specified stream, using the `DESCRIBE` method.

Other methods defined by the RTSP protocol include `SETUP`, to configure the properties of a stream; `PLAY`, to instruct the server to begin sending the stream; `PAUSE`, to instruct the server to temporarily stop sending the stream and `TEARDOWN`, to instruct

```
RTSP/1.0 200 OK
Content-Type: application/sdp
Content-Length: 5165
Date: Wed, 26 Nov 2003 16:41:42 GMT
CSeq: 1
Server: WMServer/9.0.0.3372
Last-Modified: Fri, 21 Feb 2003 18:00:22 GMT

v=0
o=- 200311261635010187 200311261635010187 IN IP4 127.0.0.1
s=Pinball WM 9 Series
c=IN IP4 0.0.0.0
b=AS:315
a=maxps:1518
t=0 0
a=control:rtsp://hmrnode00/pinball.wmv/
a=etag:{C50A55AB-C450-9F76-0DFD-8202B8AF64FC}
a=range:npt=3.000-24.833
a=recvonly
```

Figure 2.3: RTSP response

the server to end playback of the stream and release any associated resources. Other methods support setting and querying parameters, querying server capabilities and requirements, announcing the availability of a stream and requesting the server to begin recording a stream. RTSP also allows additional application specific methods to be defined. The set of methods used by an application will depend on the features provided by that application.

Responses to RTSP requests contain the RTSP version number, a status code, a reason phrase, the sequence number contained in the original request and, if appropriate, additional information specific to the request. For example, Figure 2.3 illustrates a possible response to the `DESCRIBE` request in Figure 2.2. The payload contained in the response is defined in this case by the Session Description Protocol (SDP) [HJ98].

In the case of the above response, the status code returned was `200` and the associated reason phrase was "`OK`", indicating that the request was successful. RTSP supports most of the status codes defined by the HTTP protocol. For example the `303` "`SEE OTHER`" status code can be used to inform clients that they should contact another server to perform the request. This response may be used to locate content or perform load-balancing between servers. Additional status codes specific to multimedia streaming applications are defined by RTSP. For example the `453` "`NOT ENOUGH BANDWIDTH`"

status code might be returned to a client if there is insufficient server bandwidth to supply the requested stream.

Figure 2.4 illustrates the exchange of messages between a client and two servers in a simple RTSP session. The client begins the session by submitting a `DESCRIBE` request to *Server 1*. *Server 1* responds with status code `303`, indicating that the client should submit the request to another server. The client resubmits the `DESCRIBE` request to the server indicated by the `Location` parameter in *Server 1*'s response – *Server 2* in this case. In this example, *Server 2* responds with status code `200`, indicating that the request was successful. A description of the content is contained in the response. The client requests a stream from the server by first submitting a `SETUP` request. This request will contain parameters that indicate to the server the transport mechanisms that the client can handle. The server responds in this case with the status code `200` and the parameters in the response will inform the client which transport mechanism will be used. The server begins sending the stream to the client after receiving the `PLAY` request. In this example, the client has submitted a `PAUSE` request after the entire stream has been sent to the client. The server responds with the status code `455`, telling the client that the method requested is not valid in the current state. The server's response may tell the client which methods it will allow in the current state. Finally, the client sends a `TEARDOWN` request to the server. The server releases the resources associated with the stream and sends a response back to the client.

2.2 On-Demand Multimedia Streaming Services

Multimedia streaming applications can be broadly classified into three groups, which are referred to here as *interactive*, *live* and *on-demand*.

Interactive streaming applications include, for example, video conferencing and IP telephony, and may be characterized by the presence of two or more interactive parties. All of the parties may contribute to the presentation and may or may not receive the same presentation, depending on the nature of the interaction. A *live*

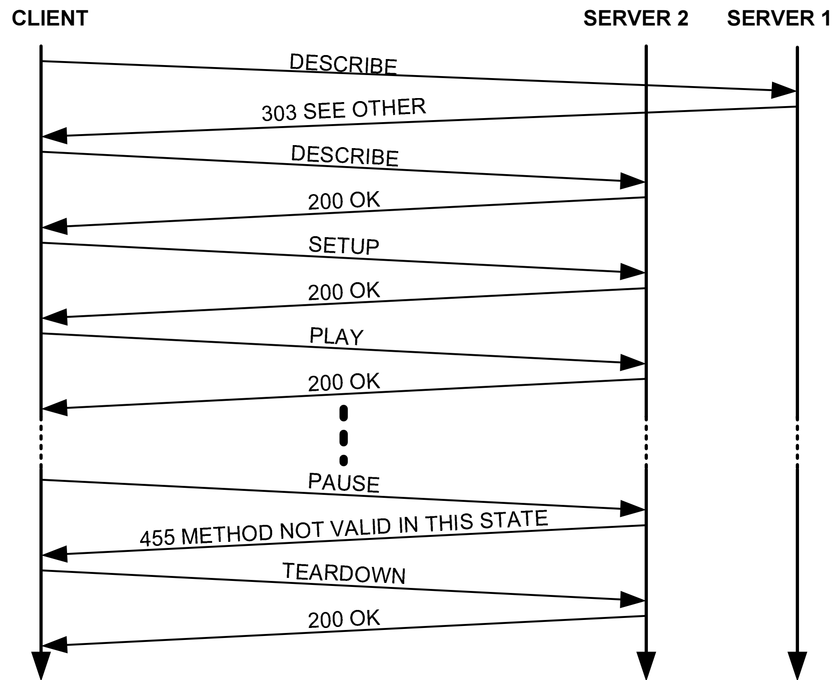


Figure 2.4: Illustration of an RTSP session

multimedia streaming service is characterized by the broadcast of a media stream to a set of clients. The clients are passive – they consume and render the stream and do not contribute to it. Clients may join or leave the stream at any time but every client sees the same presentation at approximately[2] the same time. Such a service is analogous to traditional television broadcasts.

On-demand multimedia streaming refers to a service that allows clients to request the playback of a selected presentation from an archive of stored content. The service is analogous to a World-Wide Web service that allows clients to retrieve and render web pages and embedded content, such as images. Alternatively, on-demand multimedia streaming may be viewed as an on-line movie rental store.

The flexibility of the on-demand media streaming service offered to clients can vary from one service to another. Little and Venkatesh [LV94] provide a sub-classification for on-demand streaming services. For example, a *near* on-demand streaming service,

[2]The playback of the stream at each client computer may be slightly offset due, for example, to heterogeneous client configurations or network delays.

often offered by cable television providers and in hotel rooms, usually allows clients to begin watching one of a small set of videos at fixed times and offers only very limited interactive control of the stream. Such a service can be provided by multiple parallel, temporally offset streams of the same presentation. For example, a movie may be transmitted in a number of channels, with the playback position of each stream offset by fifteen minutes. In contrast, *true* on-demand streaming offers clients the ability to choose a presentation from an archive and begin watching the presentation at a time of their choosing. A true on-demand service will usually also offer the ability to pause, resume and reposition the stream interactively, providing clients with an experience similar to renting a home movie and playing it back using a traditional DVD or video cassette player.

True on-demand multimedia streaming is significantly more demanding on server and network resources than near on-demand streaming. Both services may be provided over a computer network. A near on-demand service, however, requires only a single broadcast or multicast stream for each set of clients that started watching the same presentation at the same time. In contrast, to provide full interactive control to clients, true on-demand streaming usually requires a unicast stream for each client session. Thus, in the case of a near on-demand service, the required server and network service capacity is proportional to the number of presentations available concurrently to clients and the number of parallel, temporally offset streams of each presentation. In contrast, for true on-demand streaming, the required service capacity is proportional to the number of concurrently connected clients. As a result, designing a service to supply clients with true on-demand streaming is more challenging, since it may be difficult to predict the number of clients that will use the service concurrently and how those clients will behave.

2.2.1 Client Behaviour

The behaviour of clients in an on-demand streaming system and the resulting impact on server and network resources is an important consideration in the design and implementation of an on-demand multimedia streaming service. When designing such a service, it important to recognize that some presentations will be significantly more popular (requested with a higher frequency) than others. In other words, the distribution of request probabilities for the presentations in an archive is typically highly skewed towards the most popular ones. It has been suggested [Che94] that the *Zipf* distribution can be used to characterize the popularity distribution of requests in a video-on-demand server. This is supported by Chesire et al. [CWVL01] in their analysis of outgoing requests for media content from a university and also by Almeida et al. [AKEV01] in their analysis of the workloads of educational media servers.[3]

George Kingsley Zipf [Zip49] studied the frequency of occurrence of words in written language. He observed that, given a sequence of words ordered by frequency of occurrence, the product of the number of occurrences of a word and its rank is approximately constant for all words in the sequence. In other words, the number of occurrences of the word with rank i is proportional to $1/i$. It follows from this observation that the coefficient of proportionality is the number of occurrences of the most frequent word with rank equal to 1. More generally, according to Zipf's Law, the probability of occurrence, p_i, of a word with rank i, is given by:

$$p_i = \frac{1}{i.H_K}$$

where K is the number of different words observed and H_K is the Kth harmonic number ($H_K = 1 + 1/2 + 1/3 + \cdots + 1/K$). Another distribution related to Zipf's Law [Knu98]

[3]In the latter study by Almeida et al., the popularity distribution was described by the concatenation of two Zipf-like distributions.

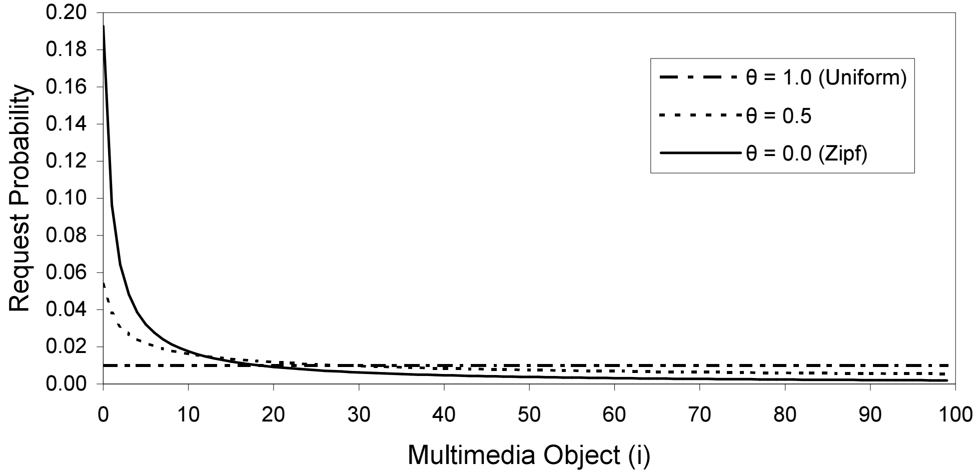


Figure 2.5: Zipf Distribution for $K = 100$ and $\theta = 0$, $\theta = 0.5$ and $\theta = 1$.

allows us to vary the amount of "skew" in the distribution:

$$p_i = \frac{1}{i^{1-\theta}.H_K^{(1-\theta)}}$$

where $H_K^{(s)}$ is the Kth harmonic number of order s, given by $1+1/2^s+1/3^s+\cdots+1/K^s$. If $\theta = 0$, the distribution corresponds to a Zipf distribution and if $\theta = 1$, the distribution is uniform with $p_i = 1/K$ for all i. Figure 2.5 illustrates the Zipf distribution for $K = 100$ and for different values of θ.

Griwodz et al. [GBW97] propose a more rigourous model to simulate client behaviour, which was developed to closely fit patterns observed in the rental of movies from a traditional movie store, but the parameters derived were specific to the data set used in the study. This study suggests that the Zipf distribution overestimates the popularity of the most popular multimedia presentations. Another more recent study, however, based on an analysis of requests to an on-demand multimedia archive, suggests that the Zipf distribution *underestimates* the popularity of the most popular presentations [AS98]. In this thesis, the Zipf distribution is taken to adequately reflect the fact that certain multimedia presentations are significantly more popular

than others. Veloso et al. [VAM+02] have shown that the behaviour of clients accessing live streamed media is fundamentally different to that of clients accessing archived presentations *on-demand*, and the scope of this thesis is restricted to the latter.

Although it has been shown that the Zipf distribution can be used to predict the relative popularity of the presentations in an archive, it cannot be used to model long-term client behaviour, where the popularity of the presentations changes over time. Dan and Sitaram [DS95b] use a "folded" Zipf distribution, which is rotated over time, to simulate gradual changes in the relative popularity of presentations. Similarly, Chou et al. [CGL00] model both gradual and abrupt changes in popularity by "shuffling" the rankings of the presentations.

Given an archive containing K presentations, each with a known bit rate, B_i, and average stream duration, W_i, and the popularity distribution for the archive, then the mean aggregate bit rate for all concurrent streams in the system can be calculated for a mean request arrival rate equal to λ. Suppose a presentation ranked by popularity in position i is requested with probability p_i, then the rate of arrival of requests for streams of presentation i is λp_i. Little's formula [Lit61], $L = \lambda W$, allows us to relate the number of streams in the system to the arrival rate and duration of each stream. Thus, for presentation i, the mean number of concurrent streams is $\lambda p_i W_i$ and the aggregate bit-rate required for this number of concurrent streams is $\lambda p_i W_i B_i$. Thus, the total aggregate bit-rate for the average number of concurrent streams of all presentations is

$$\lambda \left(\sum_{i=1}^{K} (p_i W_i B_i) \right)$$

It is worth noting that popularity alone does not determine the demand placed on a server by requests for a single presentation. A relatively unpopular presentation with a high bit-rate or long duration may consume more server resources than a more popular presentation with a lower bit-rate or shorter duration. For example, Figure 2.6 shows the server bandwidth utilization for streams of three presentations, each with the same popularity and, hence, the same mean request rate. The duration of *Presentation*

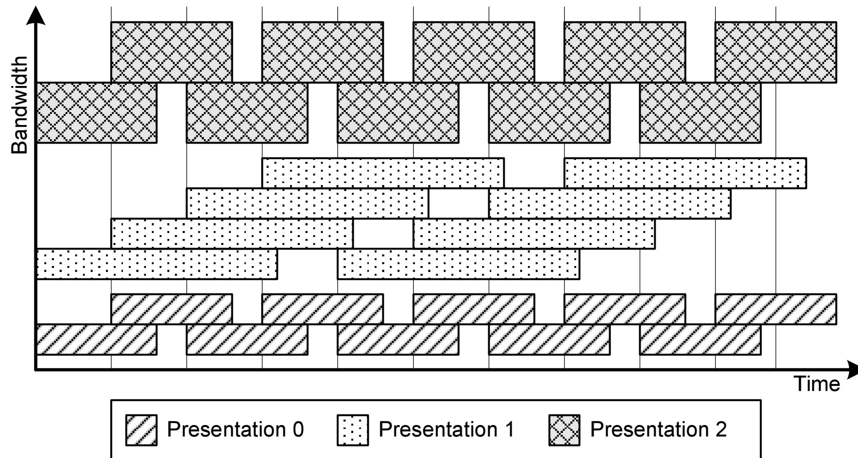


Figure 2.6: Server resource utilization for presentations with different durations and bit-rates.

1, however, is twice as long as that of *Presentation 0*, increasing the mean number of concurrent streams of the presentation and, hence, the server bandwidth utilization for the presentation. *Presentation 2* has the same duration as *Presentation 0* but has a higher bit-rate, again doubling the server bandwidth utilization for the presentation.

Intuitively, it may be assumed that the arrival rate of requests (λ in the above expression) to a multimedia streaming service will vary on a daily basis, with peaks occurring at approximately the same time each day. The time at which peaks will occur will depend on the role of the service. For example, a service used to store educational content for use in a university will probably experience a peak in request arrival rate during the morning or afternoon, whereas a public video-on-demand server containing movies will probably experience a peak during the evening. Similarly, one would expect either higher or lower daily peaks at weekends, again depending on the role of the service. This assumption is supported by studies including those of Chesire et al. [CWVL01] and Veloso et al. [VAM+02]. Although the request arrival rate, λ, may vary in this manner, the *proportion* of the aggregate server bit-rate accounted for by each presentation should remain the same, unless the relative popularity also changes if clients demonstrate different behaviour at different times of the day or week.

2.3 On-Demand Multimedia Server Design Considerations

Clusters of commodity PCs can provide a low-cost solution for implementing scalable, highly-available services to clients over the Internet [FGC+97]. For example, the Google web search engine service, mentioned in Chapter 1, is provided by clusters of over 15,000 commodity PCs and is more cost-effective than expensive, high-end server systems. [BDH03]. Performance, high-availability and scalability are achieved by the Google architecture through partitioning and replication of the search engine's data. Although the deployment of large numbers of commodity PCs will increase the frequency of node failures, the homogeneous nature of the Google application and the scalability of the architecture means that both the provision of redundant capacity and the cost of system repairs can be controlled.

To illustrate the fundamental concepts behind the server cluster model, a simple web server cluster is considered. The service provides clients with remote access to a set of static web pages and the clients retrieve individual web pages by submitting queries in the form of HTTP requests. When the service receives a request, which contains a URI identifying a web page, the page is retrieved from a storage device and sent back to the client. A simple cluster-based model to provide such a service is illustrated in Figure 2.7. The cluster model combines the resources of four commodity PCs, replicating all of the service's static web content on each cluster node. Any of the four nodes is capable of servicing any client request. When each request arrives, one of the server nodes must be selected to service the request. The aim in distributing requests among server nodes is to balance the workload across the nodes, thereby maximizing the number of requests that can be serviced. Several technologies exist to provide load-balancing functionality, including round-robin DNS [Bri95], intelligent switches capable of distributing incoming requests [Bre01] and software solutions, such as Microsoft's Network Load Balancing (NLB) cluster technology [Mic00]. If one of the cluster nodes shown in Figure 2.7 were to fail, the client requests would be distributed among the remaining three nodes. If there is sufficient redundant capacity on the

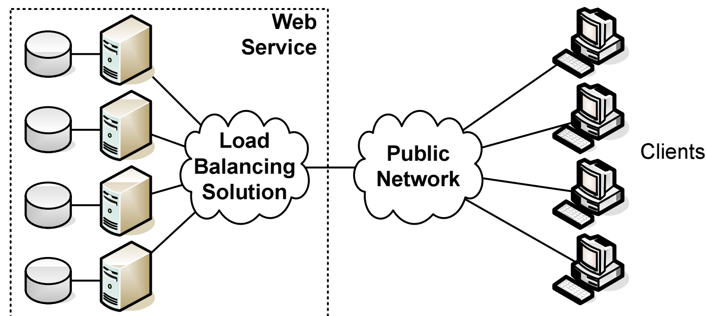


Figure 2.7: Simple four node web server cluster

remaining cluster nodes to handle the workload of the failed node, clients will not notice any degradation in service. If the service capacity of the cluster needs to be increased, to cater for an increase in client demand, additional nodes can be added, replicating the static web content on the new nodes. Scaling the storage capacity of the service without also scaling the service capacity would require additional storage capacity to be added to every node in the cluster.

It is assumed that, like the simple web server described above and other services made available to users over the Internet, the requirements for an on-demand multimedia streaming service will include scalability, high-availability and low purchase and maintenance cost, and these requirements are discussed in more detail below.

2.3.1 Scalability

Most large-scale services must satisfy two scalability requirements. First, the service architecture must be capable of meeting the expected client demand. In practice, however, it can be difficult to accurately predict client demand for an on-demand streaming service, leading to a second scalability requirement: it should be possible to increase (or decrease) the capacity of the service in response to changing client demand.

The server cluster model has the potential to satisfy both of these scalability requirements. First, the workload associated with most Internet services exhibits inherent parallelism – individual client requests can be distributed among server nodes to provide load balancing. As illustrated by the Google search engine, this allows the system

to scale to thousands of nodes. The second scalability requirement is satisfied because the server cluster model allows service capacity to be increased by adding nodes, or by replacing older nodes with new ones with better performance characteristics.

An on-demand multimedia streaming server based on the server cluster model should satisfy both of the above scalability requirements. It should be possible to implement a service to meet the expected demand and also to scale the service incrementally – by adding nodes to the cluster or by replacing existing nodes with new ones – if the actual demand differs from the expected demand. When adding nodes to the cluster, the characteristics of the nodes that are added should not need to match the characteristics of existing nodes and the process of adding new nodes should not significantly impact on performance.

2.3.2 Availability

Computer hardware and software will occasionally fail and it is important to manage such failure when it occurs, to control the reduction in service capacity. The server cluster model satisfies this requirement through node independence. The failure of any node means the overall service capacity will be reduced by the capacity of that node. Any loss of data is confined to data that only existed on the failed node and was not replicated on another node in the cluster. The remaining nodes can continue to handle requests and, if they have enough unused service capacity, can take on the workload of the failed node.

Node unavailability is not restricted to node failures and includes scheduled service "downtime". The development of new service features, maintenance of application software, installation of security patches, upgrades to operating system software and upgrades or maintenance of hardware all require nodes to be temporarily unavailable. Such downtime, although controllable, can be viewed in the same way as node failure, in terms of the impact on the availability of the service to clients [Bre01]. A large-scale cluster-based service should be designed in a manner that supports scheduled downtime to facilitate "rolling" upgrades or maintenance, where nodes are removed individually

from the cluster to perform the maintenance. When the maintenance of a node is complete, the node is reinstated and maintenance continues with the next node.

In a discussion on lessons learned from the implementation of "giant-scale" services, Brewer [Bre01] describes three availability metrics. The first of these, *harvest*, can be interpreted in different ways for different services. For services such as search engines, harvest defines the proportion of the total data set or index reflected in the response to a request. For many types of service, however, a request is either successful or unsuccessful and there is no concept of a partial response. Such services include, for example, world-wide web and on-demand multimedia streaming services. For this class of application, this thesis takes harvest to mean the amount of data available to clients as a proportion of the data that is available in the absence of any failure:

$$Harvest = \frac{\text{data currently available}}{\text{data normally available}}$$

It should be noted that harvest does not take into account the relative value of certain data items, for example, the relative popularity of web pages or of multimedia presentations.

Uptime is a frequently quoted measure of availability and quantifies the proportion of time that a component is available to service client requests. Uptime can be expressed in terms of the *mean-time-to-failure* (MTTF)[4] and *mean-time-to-repair* (MTTR) of a system's components:

$$Uptime = \frac{MTTF}{MTTF + MTTR}$$

When evaluating the availability of different multimedia server models later in this chapter, the *mean-time-to-service-loss* (MTTSL) will be considered and will be interpreted to be the mean time until *complete* service loss. This metric will take into account any data redundancy provided by the server models under examination and,

[4] Availability is often expressed in terms of *mean-time-between-failures* (MTBF) and MTTR. MTBF is simply MTTF + MTTR. MTTF is considered to be the more appropriate term [HP03].

when combined with harvest, characterizes the degradation of service when component failures occur. In other words, the MTTSL quantifies the mean time before a *complete* service failure occurs, while harvest quantifies the quality of the service provided, as the proportion of data that is available.

Uptime, as expressed above, does not reflect the relative importance of periods of uptime or downtime or the quality of service available. For example, downtime during periods of off-peak service utilization is probably not as important as downtime during periods of high utilization, in the context of many services. To reflect this, Brewer proposes the use of *yield* to reflect the importance of on-peak and off-peak times (when measured over relatively long periods) by expressing the number of client queries completed successfully as a proportion of the total number of requests submitted[5]:

$$Yield = \frac{\text{queries completed}}{\text{queries submitted}}$$

The effect of node failures on both yield and harvest is determined, at least in part, by the distribution of content among the nodes in the cluster. For example, in the simple four-node web server cluster described above, it was stated that the static web pages provided by the service were replicated on each cluster node. If a node were to fail, harvest would be maintained at 100%, but service capacity would be reduced by the capacity of the failed node. If the actual client demand exceeded the remaining available service capacity, each of the nodes would be overloaded, resulting in a reduction in yield of up to 25% (the contribution of the failed node to the aggregate service capacity of the cluster). Thus, ensuring that harvest is maintained when failures occur is of little benefit if there is insufficient redundant service capacity to cater for node failures.

Instead of replicating the static web pages in the four-node web server cluster example, the content could have been partitioned, with a disjoint set of web pages assigned to each cluster node. Partitioning the content in this way fundamentally

[5]It is worth noting that yield still does not take the value of each request into account and it may be the case that certain requests are more profitable than others to a service provider. Such metrics, however, are beyond the scope of this thesis.

changes the nature of the service. It is no longer possible to service any request at any node, and requests must be redirected to the node containing the requested web page. To simplify the discussion of the effects of node failures, it is assumed that the content is partitioned such that the load on each node is the same and that each node stores the same number of web pages. (In practice, such a partitioning of the multimedia content is likely to be infeasible.) If a single node failure were to occur, both harvest and service capacity would be reduced by 25%, with a corresponding reduction in yield regardless of the actual workload. In general, for any service with no concept of partial responses to requests, any reduction in harvest will cause a reduction in yield, with the scale of the reduction dependant on the importance of the data lost.

It may be argued that for certain classes of application, the degradation in yield for a service that partitions its data will be closer to that for a replicated service if clients are only offered the option to submit requests for available data. For example, consider a video-on-demand service supplied by a cluster of multimedia streaming servers, which is analogous to a movie rental store. Clients can select a movie from an extensive archive and have that movie streamed over the Internet to their home. If a cluster node fails, reducing the selection of movies available to clients may not deter a client from using the service. Instead, clients will be restricted to choosing from those movies that are available, as they would in a real movie rental store.

Rather than choosing either replication or partitioning of multimedia content in the provision of an on-demand streaming service, this thesis examines the use of selective replication. Important multimedia presentations can be replicated on more than one cluster node, with the less important presentations partitioned among the cluster nodes or replicated to a lesser degree, reducing any loss in harvest when failures occur and, more importantly, reducing the impact that the loss in harvest has on yield. Figure 2.8 illustrates the replication–partition trade-off spectrum.

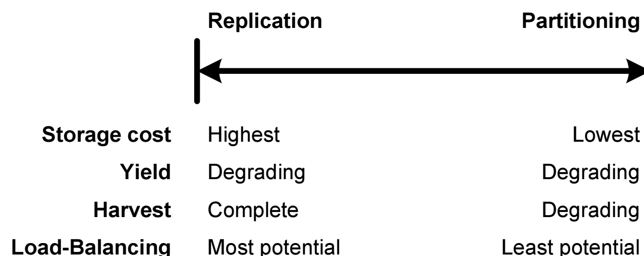


Figure 2.8: Replication–partition data distribution spectrum

2.3.3 Storage Cost

A defining characteristic of on-demand multimedia services is the volume of data that must usually be stored. For example, a two hour high quality MPEG-2 video with a data rate of 4Mbps would be approximately 3.35GB in size. A service with an archive containing five thousand such videos would require over sixteen terabytes of storage. With the rapidly decreasing cost-per-gigabyte of storage, the purchase cost of a storage subsystem with a capacity of this order of magnitude is no longer an impediment.

When implementing a large-scale multimedia on-demand service, memory, processor, network and storage interconnect resources are likely to be the constraining factors that require the use of the cluster model. To increase the overall capacity of such a service, additional nodes can be added to an existing cluster. When scaling service capacity in this way, the architectural model and, in particular, the policy used to distribute multimedia content within the server cluster, should avoid storage cost increasing proportionally with service capacity. For example, in the worst case, the implementation of a 32 node server cluster storing the archive of five thousand presentations described above will require 532TB of storage, if the entire archive is replicated on every node. Although the cost-per-gigabyte of storage equipment continues to decrease dramatically, the man-power required to manage and maintain a storage system of this size remains an important consideration [GS00], as do environmental issues such as power consumption and cooling [BDH03]. The time required to replicate a data archive of this magnitude must also be considered, as this will impact on the mean-time-to-repair (MTTR) of a node, resulting in a deterioration in the

mean-time-to-service-loss (MTTSL).

To prevent the storage capacity required by a multimedia streaming service from increasing in this manner, novel policies for distributing multimedia content among the nodes in a server cluster must be employed, and such policies are examined in section 2.4. By reducing the overall storage capacity required and making the storage system largely self-managing, the cost of operating and maintaining the service can be reduced.

2.4 On-Demand Multimedia Server Architectures

On-demand media streaming servers can be classified into one of four architectural groups, based on the method used to distribute multimedia content within the service: single-server systems, cloned server clusters, parallel servers and clusters based on dynamic replication. Each of these architectural models is described in this section and the performance, storage cost, availability and scalability of each model is discussed. In each case, the availability properties of the model are discussed in terms of harvest, yield and mean-time-to-service-loss (MTTSL). When evaluating yield, it is assumed that service capacity is fully utilized before the failure occurs. Thus, for example, a 50% reduction in service capacity will result in a 50% reduction in yield. This is the worst case and in practice there will often be spare service capacity on remaining nodes to handle some or all of the workload that would have been handled by the failed node. The potential for using storage area network (SAN) technology in multimedia streaming services is also briefly discussed, along with techniques for improving the performance of an on-demand multimedia streaming service.

2.4.1 Single-server Model

High-performance storage subsystems make it possible for a single stand-alone commodity server to supply soft real-time multimedia streams to clients over a network (Figure 2.9). However, as is the case in other application domains, this single-server

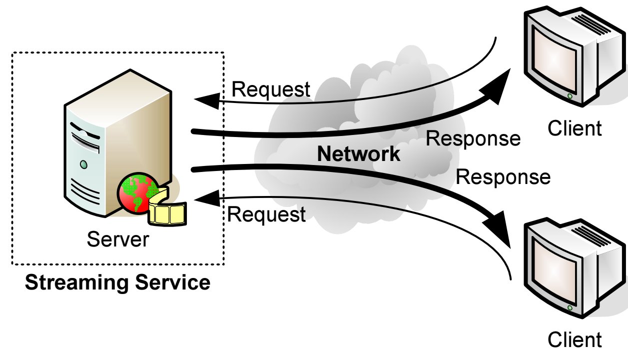


Figure 2.9: Single server model

architecture suffers from poor scalability and availability. If the number of clients increases beyond the capacity of the server, it may be possible to add additional storage devices and network adapters, but scalability is ultimately limited by the number of devices we can install in a single host. In addition, the streaming service is only as available as the stand-alone server itself. Failure of the single server will result in 100% loss of both harvest and yield. Thus, the mean-time-to-service-loss for the service, $MTTSL_{single}$, is simply the MTTF of the single server.

2.4.2 Cloned Server Model

As is the case for other applications, the scalability and availability of an on-demand multimedia streaming service can be increased by cloning the stand-alone server and replicating the entire multimedia archive on each clone (Figure 2.10), as described in the simple web server example in section 2.3. Cloned servers can be grouped into a network load-balancing cluster and incoming client requests can be distributed among the cluster nodes using a commodity network load-balancing technology. If the service workload increases beyond the aggregate service capacity of the cluster nodes, additional clones can be added, allowing the service to scale incrementally.

Harvest remains at 100% unless all nodes fail, in which case harvest reduces to zero. The MTTSL of the streaming service is therefore increased by adding cloned nodes to the cluster. To calculate the MTTSL of a cloned service, the probability that

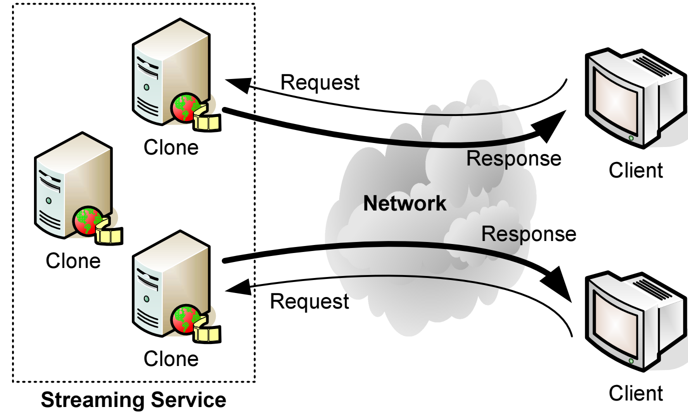


Figure 2.10: Cloned server model

the nodes remaining in a cluster after the first failure all fail before the first node is repaired must be considered. By extending the expressions for single, double and treble failures in [PGK88] and [CLG+94], we have the following expression for the MTTSL of the service, in terms of the MTTF and MTTR of the cluster's constituent nodes. The derivation of the expression can be found in Appendix A:

$$MTTSL_{cloned} = \frac{MTTF^N}{N!.MTTR^{N-1}}$$

The MTTSL of the cloned server model is significantly greater than that for a stand-alone server. For example, if each node in a four node cluster has a mean time to failure of ten days and a mean time to repair of ten hours, then the MTTSL of the service is almost sixteen years[6]. Obviously the MTTSL of the service is increased by adding more nodes. When a node fails, however, the service capacity is reduced by the capacity of the failed node. Thus, the yield after F node failures, assuming the service is fully utilized before any failure, can be expressed as:

$$Yield_{cloned} = \frac{N - F}{N}$$

[6]All MTTSL calculations in this thesis assume that node failures are independent. In practice this is not a valid assumption, as site failures due to natural disasters, power failures or software errors become more likely than large numbers of independent node failures. The expressions for MTTSL serve only to demonstrate the relative availability characteristics of each model. Correlated failures in RAID storage systems have been studied by Chen et al. [CLG+94]

It is worth mentioning that although the above expression for yield assumes that each node has the same service capacity, a cloned server cluster can be constructed using nodes with varying bandwidth and storage capacities. In this case, the storage capacity of the service is equal only to the storage capacity of the cluster node with the least capacity.

The disadvantage of the cloned server model is its high storage cost and the volume of data stored in a large multimedia archive may make this form of complete replication unattractive, as discussed in section 2.3. The high storage cost seems particularly wasteful when the frequency of requests for each presentation is considered. As described earlier, the distribution of requests for multimedia presentations is typically non-uniform and is highly skewed towards a small number of popular presentations. Requests for the remaining presentations will be relatively infrequent. To see this, consider a service that provides clients with a media archive containing K presentations, each lasting W_i seconds. Each presentation, i, is requested with probability p_i. The service is provided by a cluster of N cloned servers, with each presentation replicated on each cluster node. The mean request interarrival time is λ and the probability of assigning any given request to each server node is $1/N$. The average number of concurrent streams *on each node* of a presentation, i, is denoted L'_i and is given by:

$$L'_i = \frac{\lambda p_i W_i}{N}$$

Now consider a server cluster that contains 100 presentations replicated on four nodes. Each presentation lasts one hour and the probability that a given request is for a particular presentation is given by a Zipf distribution with $\theta = 0$. The mean request interarrival time is five seconds so each node will serve an average of 180 streams concurrently. Of these 180 streams, there will be approximately 34.2 streams of the most popular presentation, 3.42 streams of the tenth most popular presentation and only 0.342 streams of the least popular presentation on each node. Thus the first, tenth and last presentations ranked by popularity represent 19%, 1.9% and 0.19% of

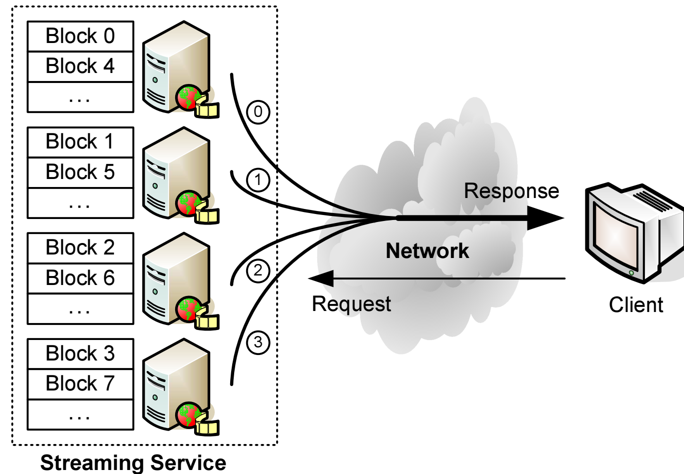


Figure 2.11: Parallel server model

each node's workload respectively. It therefore seems desirable to replicate only the most popular presentations on every node and wasteful to replicate the least popular ones, since the benefit per unit of storage capacity and the loss in yield when a node fails will be low for all but the most popular presentations.

2.4.3 Parallel Multimedia Streaming Servers

The parallel multimedia streaming server architecture [Lee98] has been widely used in the past. The concept is similar to RAID-0 [PGK88] and commercial implementations have been provided by Microsoft [BBD+96, BFD97] and IBM [HS96]. Each media presentation is divided into a number of blocks and these blocks are distributed among server nodes in a deterministic order. When a client requests a multimedia stream, the first block of the stream is retrieved from the first server node, the next block is then retrieved from the next node and so on (Figure 2.11). In this way, implicit load balancing is achieved between the nodes, while only storing a single copy of each presentation.

The availability of a parallel multimedia server is an important consideration. Unless redundant data is stored, the failure of any single node will result in complete degradation of both harvest and, as a result, yield. Thus, the MTTSL of the service

can be expressed as follows, in the terms of the MTTF and MTTR of each server node:

$$MTTSL_{parallel} = \frac{MTTF}{N}$$

For example, if each node in a four node parallel multimedia server has a mean time to failure of ten days, then $MTTSL$ of the service is only 2.5 days, which is significantly less than that for the previous two models.

Stream Reconstruction

Before a stream from a parallel multimedia server can be rendered by a player, it needs to be reassembled from the blocks stored on each server node by a *proxy*. Three proxy configurations are possible [Lee98]. In the first (Figure 2.12a), each server node reconstructs streams for a disjoint subset of clients, from both local and remote storage. This architecture has the advantage of hiding the server configuration from the clients. The processing and communication overhead, however, is high.

The second proxy configuration (Figure 2.12b) uses independent proxy servers to reconstruct streams on behalf of clients. Like proxy-at-server, this configuration also hides the underlying server configuration from clients but the processing and communication overhead is still high and the independent proxy servers and associated network infrastructure increase the cost of the system.

The final proxy configuration (Figure 2.12c) places a dedicated proxy at each client. This solution does not require additional server or network infrastructure to reconstruct client streams and the scalability of the system is improved, since each connecting client reconstructs its own stream. The disadvantage of this configuration is the need to install proxy software at each connecting client and the need for clients to know the underlying architecture and current configuration of the parallel server.

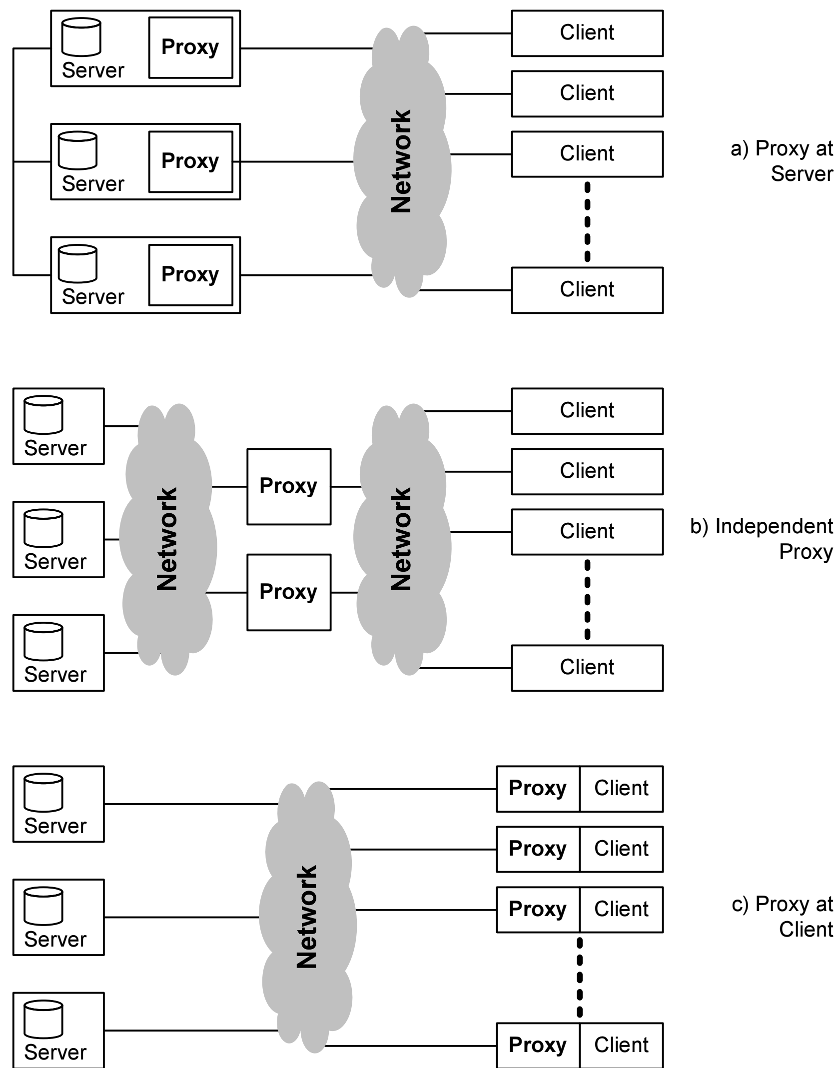


Figure 2.12: Parallel server proxy configurations

Redundancy

The issue of redundancy in parallel multimedia servers has been examined in a number of previous studies. For example, Wong and Lee [WL97] have extended the RAID concept for disk arrays to apply to server arrays. Their technique is referred to as *RAIS* (Redundant Array of Inexpensive Servers). *"Dynamic Object Striping"* allows different levels of redundancy to be applied to different media types, depending on application requirements. For example, it is suggested that videos might be stored to tolerate a single node failure, static images might tolerate two node failures and text might tolerate three node failures, allowing the service offered to clients to degrade gracefully as nodes fail. In general, a parallel server with N nodes, of which h nodes store redundant data, can continue to operate if any $N - h$ nodes remain intact. The storage overhead for such a system is $h/(N - h)$. For example, if a server with five nodes stores sufficient redundant data to operate in the presence of a single node failure, the redundancy overhead is $1/(5 - 1)$ or 25%. Similar approaches to redundancy are proposed by Bernhardt and Biersack [BB96], whose study also suggests that if the number of node failures is greater than h, playback can continue with a reduction in the playback quality of the stream. Both of the above studies propose encoding redundant data using an exclusive-OR parity approach if h is equal to one or Reed-Solomon codes if h is greater than one. Harvest remains at 100% for $0, 1, \ldots, h$ node failures but degrades completely if the number of failures exceeds h. Using a parity-based approach ($h = 1$), the MTTSL of the service can be expressed as [PGK88]:

$$MTTSL_{parity} = \frac{MTTF^2}{(N)(N-1)(MTTR)}$$

For example, if each node in a four node parallel multimedia server has a mean-time-to-failure of ten days and a mean-time-to-repair of ten hours, then $MTTSL$ of the service is twenty days, which is significantly better than a parallel multimedia server with no redundancy. Golubchik et al. have conducted a detailed comparative study of the properties of different fault-tolerance techniques in multimedia servers based on

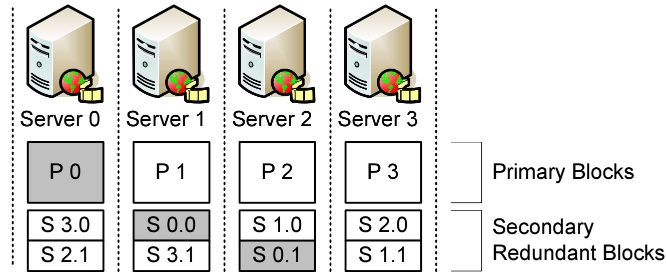


Figure 2.13: Tiger disk layout

wide data striping [GMCB01].

A different approach to redundancy is used by Bolosky et al. in the Tiger Video Fileserver [BFD97]. They considered the use of a parity-based scheme to be inappropriate, since the reconstruction of a block stored on a failed node would require every block in the stripe to be gathered to one location for reconstruction, before transmission to a client. Bolosky et al. also expected server bandwidth, rather than storage capacity, to be the most constrained resource in a multimedia streaming server and therefore proposed mirroring each stored block of multimedia content. They also proposed ‘‘declustering” each mirrored block of data, dividing the mirrored block between a number of nodes, so the work associated with supplying a missing block is shared between a subset of the remaining nodes. Figure 2.13 illustrates the disk layout used by the Tiger Video Fileserver. The shaded blocks show one of the *primary* blocks on *Server 0* with its *secondary* mirrored block declustered on *Servers 1* and *2*. In this case, the *decluster factor* is two, since each mirrored block is split between two nodes.

Using mirroring with declustering, the entire service will become unavailable if two nodes fail and the two nodes are no more than the decluster factor apart. Figure 2.14, shows a Tiger Fileserver with ten nodes and a decluster factor of four. If *Server 2* fails, as shown, the service can still supply all stored content to connecting clients, since its primary blocks are mirrored and declustered from *Server 3* to *Server 6*. Now, however, if any of the four nodes preceding *Server 2* fail, the entire service will fail since the failed *Server 2* stored some of the redundant mirrored blocks. Also, if any of the four nodes after *Server 2* fail, the service will fail, since they store the

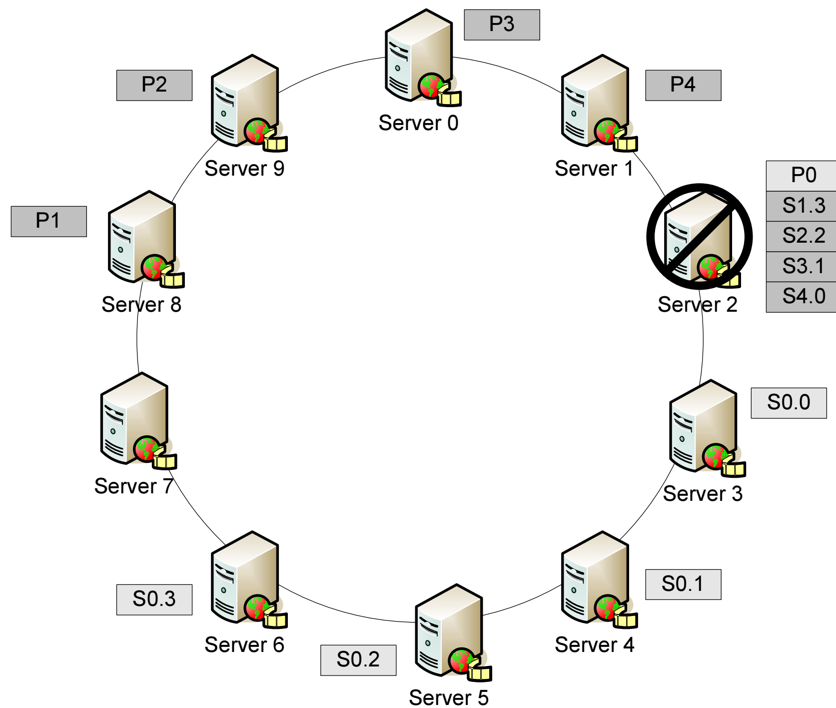


Figure 2.14: Failure of Tiger nodes

redundant mirrored blocks for *Server 2*.

Scalability

The motivation for much of the past work based on the parallel multimedia server architecture was to allow an on-demand streaming service to scale beyond the capabilities of single stand-alone servers. It has been suggested, however, that the scalability of parallel multimedia servers is limited. Bolosky et al. [BBD+96], for example, state that if the stored multimedia presentations are short relative to the number of disks, then the maximum number of concurrent streams of a presentation will be limited by the bandwidth of the disks on which parts of the presentation are stored. Chou et al. [CGL00] have compared the performance of a parallel multimedia server and a hybrid system using both striping and selective replication. Their study appears to assume a proxy-at-server architecture. (This is not unreasonable since a system designer will usually want to hide the underlying service architecture from clients.) Their

results suggest that a system based on selective replication, with even a small (10%) increase in storage capacity over an equivalent parallel multimedia server, will yield significantly better performance when the network interconnect capacity between servers is constrained.

Chou et al. also discuss qualitative disadvantages of parallel multimedia servers, which might be thought of as "practical scalability". In other words, although it may be practical to construct a system with a desired service and storage capacity, increasing the capacity of an existing service is impractical for a number of reasons. The most significant of these is the necessity to redistribute all existing multimedia content when the bandwidth or storage capacity of the service is increased by adding nodes. This is documented by a number of studies [WL97, BB96, BFD97, DS95a]. Another study [Red95] describes how the storage capacity of a parallel multimedia server might be expanded incrementally by adding N disks at a time (one to each node) to a parallel server with N nodes. Such expansion is impractical for large values of N and at some point the architecture of the server will prevent the addition of further disks. In such situations the author suggests splitting the server into two or more parallel servers, effectively creating hybrid cloned-parallel multimedia servers. Bernhardt et al. [BB96] also suggest that it may be desirable to perform striping only over disjoint subsets of server nodes, again effectively creating a hybrid architecture. The authors stated that the size of such subsets was an open research problem.

A further disadvantage of parallel multimedia servers, related to practical scalability, is node heterogeneity. With the exception of the work of Santos and Muntz on the RIO system [SM98], which relies on data replication to handle heterogeneity, a search of the existing literature describing work on parallel multimedia server architectures did not reveal any other architecture that explicitly provides for a multimedia server constructed from heterogeneous components with differing performance characteristics. Bolosky et al. [BBD$^+$96] explicitly state that the Tiger Video File Server must be constructed using identical nodes. If a parallel server were constructed using heterogeneous components, the performance and storage capacity would be limited by

the slowest or smallest component. This limitation of systems based on striping is also highlighted by Dan et al. [DS95a].

2.4.4 Dynamic Replication

The cloned and parallel multimedia server models described above both have desirable features. Both models can provide optimal load balancing across server nodes, thereby maximizing performance. The cloned server cluster model can be scaled easily by adding single nodes to existing clusters and the model also has attractive failure characteristics, with performance degrading gracefully as nodes fail and harvest remaining at 100% unless all nodes fail. The high storage cost, however, makes the use of this model less desirable for very large services. On the other hand, parallel multimedia servers have a lower storage cost but scaling existing servers is impractical and availability is poor, unless redundant data is stored. Storing redundant data incurs an additional storage cost and the availability of the service is still limited by the degree of redundancy.

This thesis examines in detail the use of selective replication, or *dynamic replication*, of content in loosely-coupled multimedia server clusters and argues that selective replication offers the best compromise between scalability, availability, performance and cost.

The model resembles the cloned server model. Instead of replicating every presentation on each node, however, a non-disjoint subset of the presentations is stored on each node. Client demand for individual multimedia presentations is periodically estimated and this information is used to determine the membership of the subsets, which are chosen to approximate an even distribution of load across all nodes. Some of the presentations are stored on more than one node to facilitate load-balancing, satisfy high demand for individual presentations or increase service availability (by reducing the impact on harvest of node failures). The relative popularity of multimedia presentations changes over time, so the assignment of presentations to nodes must be continuously or periodically reevaluated. As with the cloned server model, each node

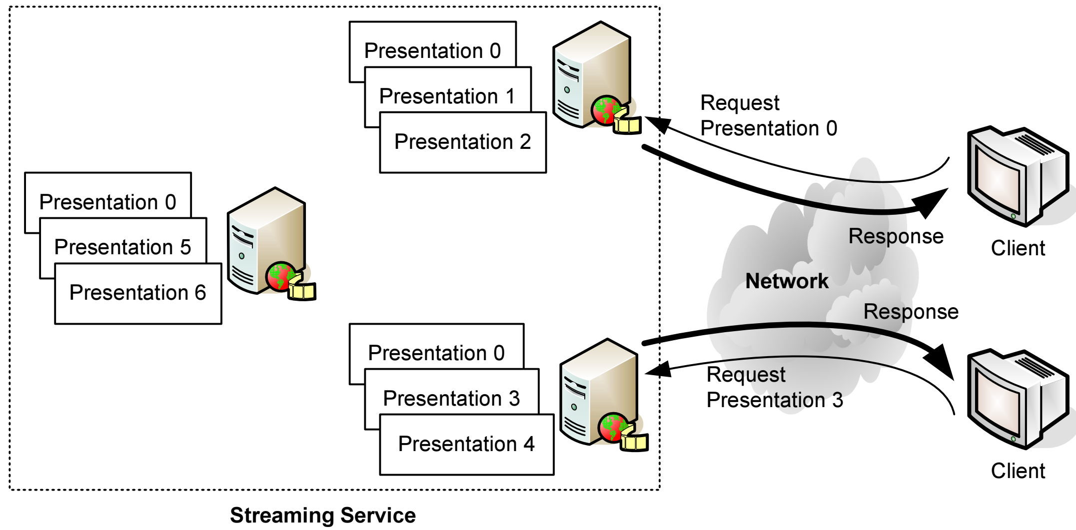


Figure 2.15: Dynamic replication server model

can independently supply streams of the presentations stored on that node to clients. It important to note that the use of dynamic replication in a server cluster environment does not prohibit the use of striping across the disks attached to each individual node, creating a hybrid replication-striping architecture. In this case, a stripe group could be treated as a single logical storage device attached to a cluster node, as proposed by Chou et al. [CGL00].

Figure 2.15 illustrates the dynamic replication model. In the example, *Presentation 0* is replicated on all three nodes and the remaining presentations are distributed among the nodes. The distribution of presentations is such that the expected service workload will be distributed evenly across all nodes, based on the estimated demand for each presentation. When a client requests *Presentation 0*, the stream can be supplied by any node with sufficient available bandwidth. Requests for any of the remaining presentations can only be handled by the node with a replica of the requested presentation.

When distributing multimedia presentations among nodes, the aim is to avoid a situation where the only node or nodes with a replica of a requested presentation do not have sufficient available service capacity to supply the requested stream, while

sufficient capacity exists on another node without a replica of the presentation. If no node has sufficient capacity to service the stream, regardless of the location of the requested presentation's replicas, the rejection of the request is unavoidable regardless of the architectural model used.

Harvest may decrease as cluster nodes fail. In the worst case and assuming each node stores the same number of presentations, the harvest for a cluster of N nodes, F of which have failed, can be expressed as:

$$Harvest_{replication} = \frac{N - F}{N}$$

The degradation in harvest is reduced as more presentations are replicated on more than one node. For example, in the simple case illustrated in Figure 2.15, if the leftmost server node failed, only *Presentation 5* and *Presentation 6* would become unavailable. Thus, harvest is highly dependent on the replication policy implemented by the server.

On first inspection, the mean-time-to-service-loss of a multimedia streaming service based on the dynamic replication model is the same as that for the cloned model:

$$MTTSL_{replication} = MTTSL_{cloned} = \frac{MTTF^N}{N!.MTTR^{N-1}}$$

If the mean-time-to-repair for a node in both models is considered, however, the dynamic replication model benefits from only storing approximately $1/N^{\text{th}}$ of the multimedia archive, whereas the cloned model required each node to store the entire archive, increasing the MTTR and, as a result, decreasing the MTTSL by a factor of N^{N-1}. As additional replicas of presentations are created to increase the availability of those presentations, the difference between the MTTSL of the two models will be reduced.

If service capacity is fully utilized when nodes fail and the workload is ideally distributed across all nodes, then yield will degrade in the same manner as the cloned server model. Otherwise, if the service capacity is not fully utilized, there will be some reduction in yield as a result of any reduction in harvest. If there is sufficient time between node failures, however, services based on dynamic replication can recover and

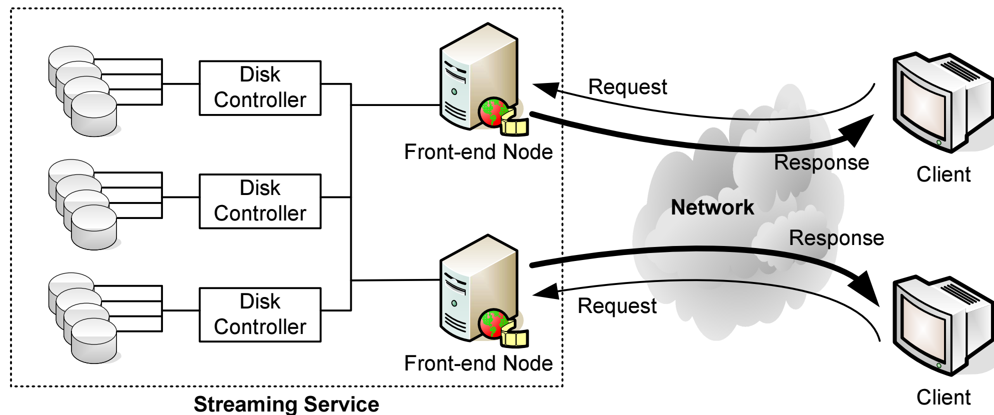


Figure 2.16: Storage Area Network (SAN) architecture for on-demand multimedia streaming

re-replicate the more popular presentations between node failures. This behaviour is demonstrated for a prototype server cluster in Chapter 6.

2.4.5 Storage Area Networks

The use of Storage Area Network (SAN) technology in multimedia servers has been investigated in the past [Guh99]. The approach that has been suggested is to provide a cluster of front-end multimedia streaming servers with shared access to SAN storage devices (Figure 2.16), thus separating the storage of multimedia content from the provision of client streams. Client requests can be distributed among the front-end nodes to provide load-balancing and each node retrieves required multimedia content from the SAN storage devices. In this architectural model, however, the issues with performance, scalability, availability and cost are merely moved from the front-end nodes to the back-end storage devices, which must implement a content distribution policy similar to those used by the cloned, parallel or dynamic replication server models [BSS00]. For example, the PRESTO multimedia storage network [BBS99] uses wide data striping across multiple stripe groups to perform load-balancing in a storage area network. The use of SAN technology also raises the possibility of implementing a storage hierarchy using storage devices other than magnetic disks, such as optical or tape devices. Such hierarchies are beyond the scope of this thesis.

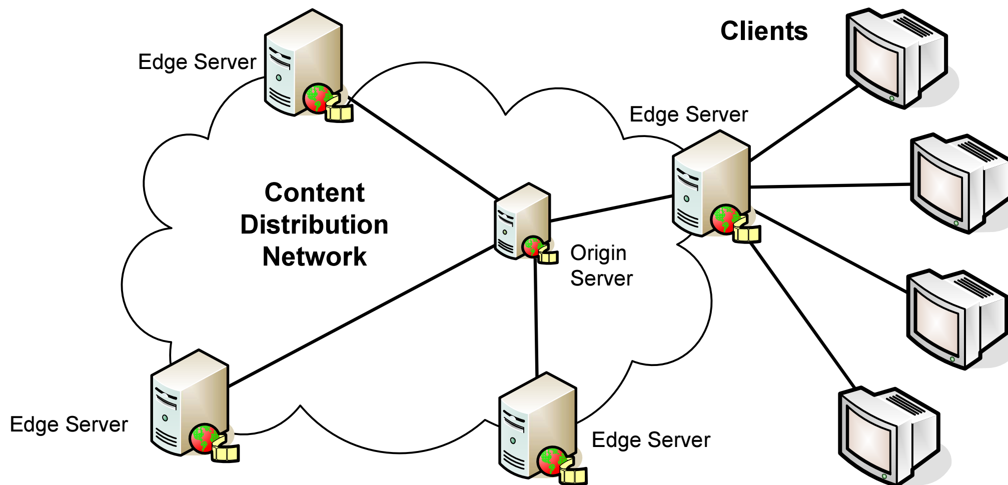


Figure 2.17: Content distribution network

2.4.6 Content Distribution Networks and Caching Proxy Servers

When large numbers of widely distributed clients access content available only at single site, there is the potential for the network infrastructure surrounding that site to become congested. To avoid such congestion and improve the responsiveness of the service, many applications – particularly those serving largely static content, such as web pages, web-based advertisements or on-demand multimedia archives – distribute their content at remote sites or *edge-servers*, allowing client requests to be redirected to sites other than the original content publisher or *origin server*. The network of origin servers and edge-servers is referred to as a *content distribution network* (CDN) [Ver02]. Figure 2.17 illustrates the CDN concept.

Frequently, the edge- or origin-server at a CDN site will be implemented as a cluster of servers, to allow the site to scale over time and to increase the availability of the site. The protocols and policies required for the distribution of content and redirection of client requests between sites in a CDN and between servers at a single CDN site are usually quite different [Ver02]. For example, the redirection of a client within a CDN might be driven by a requirement to reduce client response times, whereas redirection of a request to a particular server at a single site might be driven by the need to maximize the service capacity of the site. The scope of this thesis is restricted to

those issues relating to the architecture of a single site. Despite this restriction, however, the prevalence of CDNs requires that they be considered in the design of a on-demand multimedia streaming service for a single site. For example, Cranor et al. [CGK+01] describe the PRISM CDN architecture for capturing live broadcast streams at edge servers. To coordinate the distribution of content and perform resource management across the CDN, PRISM requires edge-server sites to report content usage statistics and current site load to content managers. Accordingly, each site in a CDN based on PRISM should be capable of providing this information regardless of whether the site is implemented as a single server or as a cluster of servers.

Multimedia caching proxy servers exploit temporal locality of reference, among other techniques, to reduce network congestion caused by multimedia streaming and to reduce network jitter. Examples include the work of Acharya et al. [AS00], which proposes a distributed caching system in which client hosts collaborate to provide cache storage space, and the work of Sen at al. [SRT99] which caches the beginning of streams to reduce latency and network jitter. Distributed caching schemes such as these are beyond the scope of this thesis.

2.4.7 Multicast Stream Delivery

The use of multicast multimedia streams has been proposed in the past as a method of reducing the network and server resource requirement for streams of the same multimedia presentation. Specifically, multicast multimedia streaming techniques attempt to send each packet of a multimedia stream just once over each branch in a network within some time window [MS02]. The basic scheme is referred to as *batching* [DSS94]: when a server receives a request to begin a new stream, it delays its response for a short time and any further requests for the same presentation, which arrive during the delay, are serviced with a single multicast stream. This scheme has obvious disadvantages, including the delay in starting new streams and the inability to provide full interactive controls, such as fast-forward and rewind. These issues are addressed by Hua et al [HCS98] and Liao et al. [LL97], among others. Again, such techniques are

beyond the scope of this thesis but may complement suitable multimedia server cluster architectures.

2.5 Summary

This chapter has introduced multimedia streaming and described the protocols required to transmit multimedia streams across a network. A true on-demand multimedia streaming service supplies many such streams to connected clients, giving each client a stream that can be started at any time and controlled independently of other streams. The provision of such a service usually requires a unicast stream for each client, making true on-demand streaming particularly demanding on server and network resources. Techniques such as multicast batching or the use of content distribution networks (CDNs) may alleviate the demand on resources.

The popularity of the multimedia presentations made available by an on-demand streaming service tends to be highly skewed, with a small number of presentations receiving the majority of requests. The Zipf distribution has been shown by various studies to yield a good approximation of the relative popularity of presentations over the short term. The relative popularity of presentations also changes over the lifetime of each presentation. The resources required for each multimedia presentation in an archive is a function not only of popularity, but also of the bit-rate and average duration of streams of the presentation, and it is important to consider these factors when designing an on-demand multimedia server.

Providing a true on-demand multimedia streaming service on a large scale requires an appropriate architectural model. The server cluster model is widely used in the implementation of large scale Internet services, such as World-Wide Web servers. Full replication of all available content to optimize performance, however, is often impractical for multimedia servers because of the large quantity of data that would need to be replicated, the resulting purchase and management cost of suitable storage subsystems and the time required to replicate a large multimedia archive when adding or

replacing nodes, or recovering from failures. Parallel multimedia servers also achieve near optimal performance, but at the expense of scalability and availability.

In the next chapter, existing work on selective dynamic replication will be discussed. Chapter 5 describes the HammerHead multimedia server cluster architecture, which is based on the use of dynamic replication and, specifically, the Dynamic RePacking content replication policy proposed in Chapter 4. Results presented in Chapter 6 will demonstrate that the cluster architecture and dynamic replication policy proposed by this thesis can approach the performance of a cloned server cluster, while requiring significantly less storage capacity. The results will also demonstrate the impact on performance of reductions in harvest.

Chapter 3

Dynamic Replication and Group Communication in Multimedia Server Clusters

This chapter describes the role of a dynamic replication policy – used to perform selective replication of content in a multimedia server cluster – and discusses the existing dynamic replication policies that have been previously proposed.

The HammerHead architecture proposed by this thesis makes extensive use of group communication. Informally, group communication systems provide a means to create groups of distributed processes, in which each member of the group consumes a common stream of messages sent to the group. For example, IP multicast can be considered a form of group communication, albeit one that provides no guarantees about message delivery. Group communication is discussed in greater detail in Chapter 5, but this chapter presents a brief overview of previous uses of group communication in the implementation of multimedia server clusters.

Terminology

The term "*replica*" is used throughout this thesis to refer to a stored instance of a presentation and a presentation that is described as having one replica is stored on just

one cluster node.

3.1 Dynamic Replication Policies

In general, for any multimedia streaming service based on dynamic replication, in which each node stores a subset of the presentations made available by the service, the distribution of the replicas of each presentation among the cluster nodes may be described by an assignment matrix, A. If there are N nodes and K unique presentations, then A is an $N \times K$ matrix and the element $a_{i,j}$ has the value 1 if node i has a replica of presentation j, and has the value 0 otherwise [WYS95]. The following assignment matrix, A_t illustrates an assignment of fifteen presentations to four nodes at time t, with the columns corresponding to presentations and the rows corresponding to nodes.

$$A_t = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \tag{3.1}$$

If the matrix A_{t+1} describes the new assignment of replicas to nodes after reevaluating the distribution, then the matrix $A_{t+1} - A_t$, denoted A'_{t+1}, describes the changes that must be made to implement new distribution of replicas. If $a_{i,j} = 1$, then a new replica of presentation j should be created on node i and, if $a_{i,j} = -1$, then the replica of presentation j should be deleted from node i. Otherwise, if $a_{i,j} = 0$, no change is required. The following example illustrates an assignment matrix, A_{t+1}, corresponding to a reevaluation of the assignment in the example above.

$$A_{t+1} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \tag{3.2}$$

The changes between the two assignments illustrated above are described by the following matrix:

$$A'_{t+1} \begin{pmatrix} -1 & +1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & +1 & -1 & 0 & +1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & +1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & +1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.3}$$

Thus, given some assignment matrix A_t, the function of a dynamic replication algorithm is to generate a new assignment matrix, A_{t+1}, and hence the matrix A'_{t+1}. If the new assignment matrix, A_{t+1}, is based on the preceding matrix, A_t, then the policy is incremental, otherwise, each invocation of the algorithm is independent.

The primary goal of a dynamic replication policy is to be able to service any client request for a presentation stored on the cluster, as long as there is sufficient service capacity for the stream on *any* node in the cluster. Obviously this goal will be achieved by a cluster of cloned servers, since every node will have a replica of every presentation. However, when dynamic replication is used, the multimedia archive is partitioned and selectively replicated. With each presentation stored on a subset of the cluster's nodes, it is possible that a request may be rejected due to insufficient service capacity on those nodes with a replica of the presentation, while another node has sufficient service capacity but no replica.

Little and Venkatesh [LV95] proposed a justification for the conjecture that the probability of a client request being successfully serviced is maximized when replicas of presentations are assigned to disks (or nodes, in the case of a multimedia server cluster) such that the probability of accessing each disk is the same, subject to the constraints that both nodes and presentations have heterogeneous storage and streaming characteristics. It was also noted by Little and Venkatesh in the same study that assigning movies to disks to balance disk access probabilities, while minimizing storage utilization, is an NP-hard bin-packing problem, but that more efficient approximations using heuristic approaches can be shown to yield good performance. Sitaram et al. [SDY93]

also argue for this approach in the context of general purpose file servers and the approach has been adopted by most of the algorithms described below, as well as by the Dynamic RePacking algorithm proposed in Chapter 4.

DASD Dancing

The DASD Dancing scheme [WYS95] proposed by Wolf et al. consists of two algorithms: a dynamic algorithm, which attempts to transfer active client streams from overloaded cluster nodes to under-loaded ones and a static algorithm, which aims to assign presentations to nodes in a manner that is synergistic with the dynamic algorithm. The static assignment algorithm can operate in both incremental or "from scratch" modes.

The dynamic component of the DASD Dancing scheme operates on a directed graph, G, of the cluster's nodes. The graph G, contains a directed arc from a node i_1 to another node i_2 if both nodes have a replica of at least one common presentation j, if there is at least one stream of j being served from i_1 and if there are sufficient resources available to serve a stream of j on i_2. The algorithm is initiated when the load imbalance between cluster nodes exceeds some defined threshold and the algorithm proceeds by repeatedly transferring active streams from over-loaded nodes to under-loaded nodes, along the shortest path in the graph G, until either no further stream transfers are possible or the cluster load imbalance no longer exceeds the defined threshold.

The static "from scratch" component of the DASD Dancing scheme uses the expected demand for each presentation to evaluate a suitable assignment of replicas to nodes. The algorithm begins by determining the required number of replicas of each presentation using an apportionment method – specifically Webster's monotone divisor method. Such apportionment methods are frequently encountered in politics when determining a fair allocation of representatives to states or parliamentary constituencies. The DASD Dancing scheme assigns each presentation a required number of replicas, based on the expected demand for each presentation and the available storage capacity of the cluster. Once the assignment has been made, the most demanding presentations

will be assigned multiple replicas, while the remaining less demanding presentations will be assigned just a single replica. The static assignment algorithm proceeds by first assigning those presentations with more than one replica to nodes in order to achieve high connectivity in an undirected graph, H, with an arc between nodes that store replicas of common presentations. The remaining single replica presentations are then assigned to nodes using a greedy algorithm which balances the expected load on the cluster's nodes, subject to the constraint that there is sufficient storage to store a replica on a selected node. In the static algorithm's incremental mode of operation, which is executed periodically, Webster's monotone divisor method may dictate that some presentations should lose replicas while other presentations will gain replicas. Replicas are removed where necessary such that there is a minimal increase in the diameter of the graph H. Similarly, replicas are added where necessary to reduce the diameter of H.

The performance of DASD Dancing was originally evaluated using simulation experiments and was shown to perform well in relation to static schemes that do not migrate active streams between nodes. The results of the simulation also show, however, that when the dynamic component of DASD Dancing is combined with a greedy "Least Loaded First" static assignment scheme also described in [WYS95] – which makes no attempt to reduce the diameter of the graph H – there is still a significant improvement in performance over a scheme without DASD Dancing's dynamic component. This suggests that the dynamic component of DASD Dancing may be used to improve the performance of other static schemes and does not rely on the static component of the scheme.

The static component of the DASD Dancing scheme leaves a number of practical issues unresolved. For example, consider the situation in which presentations are to be assigned to a cluster with significantly more available storage capacity than is required. In this case, Webster's monotone divisor method will assign a large number of replicas to each presentation and in extreme but perhaps not unusual cases, will result in a replica of every presentation being assigned to every node producing a cluster of cloned multimedia servers. While it might be argued that such cloning should be performed if

there is sufficient storage capacity, the time required to replicate every presentation on every node, particularly when recovering from failures, should be taken into account.

It is not clear how DASD Dancing would handle the migration of streams of presentations with different bandwidths in a single iteration of the dynamic algorithm. The addition and removal of nodes to and from a cluster and the removal of nodes due to failure has also not been addressed. Unlike DASD Dancing, this thesis makes no assumptions about client and server support for server-initiated migration of active streams between streaming sources, without interrupting playback.

Optimized Replication and Placement

Zhou and Xu [ZX02a] propose a scheme to determine an optimal number of replicas of each presentation in an archive. Like the static component of the previous DASD Dancing scheme, this scheme uses an apportionment algorithm (Adams' monotone divisor method). Unlike DASD Dancing, however, the number of replicas created for each presentation is bounded by the number of nodes in a cluster. To reduce the complexity of the apportionment algorithm, the authors have proposed an approximation that takes advantage of the expected Zipf distribution of popularity. Once the number of replicas of each presentation has been determined, a *smallest-load-first* algorithm is used to assign replicas to nodes such that there is an upper bound on the expected load-imbalance between the nodes.

Each invocation of the proposed replication and placement scheme is independent and the authors do not address the cost of adapting to changes in the relative popularity of presentations. In addition, it is assumed that each presentation has the same duration and bit-rate and that each node has the same network bandwidth and storage capacity. Although the authors propose a scheme to handle varying bit-rates and constrained storage conditions, the scheme appears to assume that the replica placement policy can modify the bit-rate of stored presentations.

The same authors have also proposed an architecture to improve the performance of a multimedia server cluster based on selective replication [ZX02b]. In the proposed

architecture, a “backbone” cluster network is used to allow a node without a replica of a requested presentation to obtain the stream from another cluster node, before sending it to the client. An architecture such as this would complement the HammerHead cluster architecture described later in this thesis, improving the performance of a services based on dynamic replication, at the cost of providing the backbone cluster network.

Bandwidth-to-Space Ratio (BSR)

A logical storage device in a multimedia server cluster, for example, a single disk or an array of disks, has a fixed I/O bandwidth and a fixed storage capacity. The ratio of these two quantities can be expressed as the bandwidth-to-space (BSR) ratio of the device. Similarly, the load on a device resulting from the multimedia streams it is expected to supply and the total used storage capacity of the device can be expressed as the BSR of the content stored on the device. The bandwidth-to-space ratio policy proposed by Dan et al. [DS95b] assigns replicas of presentations to logical storage devices in a manner that attempts to reduce the deviation between the BSR of a device and the BSR of the replicas stored on the device. This is done to avoid the situations where a device with no spare storage capacity has available bandwidth and a device with no spare bandwidth still has available storage capacity. The allocation of replicas to storage devices is based on an estimation of the bandwidth required to service the expected number of concurrent streams of each presentation.

The BSR policy proceeds as follows. Each presentation is processed in turn, with the best results obtained when the presentations are processed in decreasing order of demand. The assignment of each presentation to one or more storage devices takes place in phases, with the first two phases only applying to those presentations whose demand has increased. In the first phase, the policy evaluates whether the current replicas are sufficient to satisfy the expected demand for the presentation. This evaluation is based on each storage device with a replica of the presentation being able to contribute the free bandwidth of the device, up to a system wide limit called the *availability parameter*, to service the expected additional load. If the first phase dictates that additional replicas

are required, the second phase creates those replicas on nodes whose deviation between device and stored content BSRs would be reduced by storing the replica. Devices are considered in order of decreasing deviation. In the third phase, the expected load is reallocated among stored replicas in order to reduce the BSR deviation of the devices. In the final phase, if additional load could not be allocated, the policy attempts to remove replicas if their demand can be satisfied through reallocation of expected load from one replica to the other replicas of the presentation. This will increase the allocated bandwidth on devices that have exhausted their storage capacity, while freeing both storage and bandwidth on other devices.

The literature describes an evaluation of the BSR policy using a simulation study, which generated the expected demand for videos from a Zipf distribution and used the generated demand as the input to the policy. An actual client workload was not simulated so there are no experimental results to evaluate how well the algorithm performed, based on the estimation of the demand for each video. It is not clear from the literature how the availability parameter – which will have a significant impact on the number of replicas generated for each presentation and the overall performance of the server – is chosen. The availability parameter is applied on a system-wide basis, and the use of different availability parameters for different presentations has not been investigated. Changing the availability parameter in this way would allow the balance between replication and partitioning of multimedia content to be controlled, as discussed in Chapter 2.

Greedy Pseudo-Allocation

Venkatasubramanian and Ramanathan [VR97] propose a "pseudo-allocation" algorithm that assigns replicas of presentations to cluster nodes based on a greedy heuristic. The algorithm begins by constructing a placement matrix, PM, with rows corresponding to presentations and columns corresponding to cluster nodes, similar to the assignment matrix presented earlier in the chapter. Each entry $PM_{i,j}$ contains the maximum number of streams of a presentation j that could be serviced if a replica of the presentation

was placed on a node i – based not only on network bandwidth, but also on CPU, memory and disk bandwidth resources – up to a maximum of the number of expected requests for the presentation over a period of time. This value is scaled by a value that quantifies the *revenue* associated with the replica, which reflects the resources required by a stream of the presentation, the presentation's popularity and other characteristics such as the charge to a client for receiving a stream of the presentation. The algorithm repeatedly evaluates the maximum element in the matrix, $\max(PM_{i,j})$, and assigns the corresponding presentation i to node j. Each matrix element is reevaluated taking this assignment into account and the procedure is repeated. This greedy approach will tend to have the effect of creating fewer replicas for the presentations that have the highest resource requirements, whereas intuition suggests that more replicas should be created for the most popular or demanding presentations to increase the ability to perform load-balancing.

As well as the pseudo-allocation algorithm for assigning presentations to nodes, Venkatasubramanian and Ramanathan propose an adaptive scheduling scheme, which creates an additional replica of a presentation when a request that cannot be serviced immediately arrives at the server. This adaptive scheme fulfills a similar role to the dynamic segment replication policy proposed by Dan et al. [DS95a], which is described later in this chapter.

Predictive Fault-Tolerant Placement

A scheme for performing predictive fault-tolerant placement of replicas in a multimedia server cluster, based on a development of the greedy pseudo-allocation scheme described above, has been proposed by Wei and Venkatasubramanian [WV01]. Under this scheme, each presentation is assigned a required *replication degree*, dictating the number of replicas of the presentation required for fault-tolerance. The pseudo-allocation algorithm is executed as described above, after which some presentations may have satisfied their replication degree, some may have fewer replicas than the replication degree, some may have more replicas than their replication degree and some may have no replicas. The

fault-tolerant phase of the algorithm attempts to enforce the replication degree of each presentation by creating additional replicas on nodes that do not already contain a replica of the presentation. This fault-tolerant phase of the assignment algorithm does not take load-balancing into account when assigning replicas to nodes, as the Dynamic RePacking policy proposed later in this thesis does. Additional replicas of presentations may be placed either on random nodes (subject to storage constraints), or within groups of nodes, making the reliability of the service (specifically, the loss in harvest, rather than yield) more predictable.

Threshold-Based Replication

The dynamic replication policies described so far all either estimate the future demand for each presentation or assume that such information is available from an external source. Lie et al. [LLG98] use a threshold-based approach to continuously monitor the resources available for supplying streams of each presentation and trigger replication when necessary. When a request for a stream of a presentation j arrives, the server evaluates the available service capacity – measured in units of streams – for servicing requests for j. This available capacity is the sum of unused service capacity on all nodes with a replica of j. If the available service capacity is below a threshold, and if the presentation is not already being replicated, then a replica is created on a new node. The threshold changes dynamically and is equal to the expected number of new requests for the presentation that will arrive before a new replica can be created. If the threshold exceeds a configurable fraction of the average service capacity of a node, then this latter quantity is taken as the threshold, to prevent excessive replication. The evaluation of the replication threshold would require knowledge of the popularity of the presentation, to allow the number of future requests to be estimated. A related work [CGL00] suggests using the last inter-arrival time for the presentation in place of the popularity. When the replication threshold is exceeded, a suitable node is selected to store the new replica and replication of the presentation begins. A similar threshold-based scheme is used to trigger the removal of replicas. A threshold-based replication

scheme such as this would delay replication until server load is high, whereas the other dynamic replication schemes described in this chapter assign presentations to nodes based on expected demand, which may be estimated during periods of lower server load, allowing the assignment to take place during off-peak hours.

The same study proposes three schemes for creating new replicas, once a suitable assignment of presentations to nodes has been determined: a sequential replication scheme, in which a new replica is created by streaming the presentation from a source node to a target node; a piggyback scheme, in which the replication stream is piggybacked on a client stream; and a parallel replication scheme, in which portions of the presentation are replicated using multiple parallel streams from different replicas. A related study [CGL00] proposes the use of "early acceptance" when creating a new replica of a presentation: during the creation of the replica, new client streams can be serviced by the partially created replica. Such a scheme, however, assumes that the new replica will be created successfully in a timely fashion, that clients won't fast-forward to a nonexistent portion of the presentation and that the multimedia streaming protocols and file formats used support streaming of partial replicas. Each of these schemes for creating new replicas are independent of the threshold-based method for triggering the addition and removal of replicas and could be used with other dynamic replication algorithms.

Dynamic Segment Replication

Unlike the other replication policies described previously, the *Dynamic Segment Replication* policy proposed by Dan et al. [DS95a] aims to counteract transient overloads on logical storage devices by temporarily copying a segment of a replica to a lightly loaded storage device. The policy may be used in conjunction with a replica assignment policy, such as those already described. The policy relies on the sequential nature of multimedia streaming – in the majority of cases, multimedia files are read sequentially from beginning to end, making it possible to predict the data that will be read in the near future with a high probability of success.

The original work was based on a single server system with multiple logical storage devices and suggested that temporary segments could be created by writing the segment as it is read into memory before being sent to clients. In a loosely-coupled cluster architecture, the creation of the temporary segments would place additional load on already overloaded cluster nodes.

3.2 Group Communication in Multimedia Server Clusters

A highly-available, fault-tolerant video-on-demand service, based on the use of group communication, has been described by Anker et al. [ACK+97, ADK99]. This work was later generalized by Fekete and Keidar [FK01] to consider the implementation of highly-available services using group communication, and the work of Anker et al. was presented as an example. The framework assumes that a service is provided by a set of cluster nodes and that clients interact with the service in sessions. In each session, a client and a single cluster node exchange request and response messages, but not necessarily in request-response pairs. If a client disconnects and reconnects, one session ends and a new session begins. Like the HammerHead architecture described in this thesis, the framework assumes that each node stores a subset of complete *content units* (for example, video presentations in the case of a video-on-demand service), that the content units are read-only and may be replicated on a subset of the cluster nodes to increase availability, and that clients interact with a single node during a session.

Like the HammerHead architecture proposed in this thesis, the work of Anker et al. and the related work of Fekete and Keidar is based on a cluster of nodes using commodity hardware and network technologies. The nodes coordinate their actions through the use of group communication and in the specific case of the video-on-demand service described by Anker et al., the Transis group communication system [DM96] is used.

In the framework proposed by Fekete and Keidar and the video-on-demand service of Anker et al., cluster nodes and clients dynamically join and leave overlapping groups

of processes. The system implements three types of process group. There is a single *server group* containing every node in the server cluster. For each content unit stored on the cluster, there is a *content group* containing those cluster nodes with a replica of the unit. Finally, for every client session, there is a *session group* containing the client, the cluster node servicing the session, and a number of session backup nodes. Connecting clients initially communicate with either the server group (in the case of the video-on-demand service) or a content group (in the case of the generic framework) to request a new session. The session group is used by clients to send requests to the service. The backup nodes in a session group (which are also members of the content group for the content unit being supplied) observe requests sent by the client to the node servicing the client's requests, allowing the backup nodes to maintain state information for the session. Should the node servicing the requests fail, one of the backup nodes can resume servicing the requests, using this state information. The nodes in a content group continuously update each other with information describing each session and this information is used to implement load-balancing. In the case of the video-on-demand service, the information includes the playback position of each stream and the playback rate, and the information is updated periodically (e.g. every 0.5 seconds).

When a cluster node fails, the other members of each of the content groups of which the node was a member will be informed of the failure by the group communication service. For any given content unit, the associated sessions will be evenly distributed among the members of the group. Similarly, if a new node with a replica of the content unit joins the cluster, the existing load will be redistributed within the new content group. The changes resulting from any redistribution in load may result in changes in the membership of any session groups.

The work described by Anker et al. and Fekete and Keidar is the closest existing work to the HammerHead architecture described in this thesis, but differs from HammerHead in a number of important ways:

- The existing work assumes that each video is stored on a subset of the cluster nodes, but does not address the policy used to assign videos to nodes or its

implementation, as this thesis does. Instead, the focus is on the detection of node failure, dynamic rebalancing of workload when nodes join or leave a content group group, and transport protocol and buffering issues when a new cluster node takes over serving an existing stream.

- Unlike the existing work, HammerHead makes no assumption about the ability to perform server-side recovery of multimedia streams when nodes fail. This decision was made to allow the HammerHead service to support existing off-the-shelf media players and to reduce the size and frequency of updates to the streams' session state.

- The communication channel used by clients to control a session and, in the case of the video-on-demand service, provide flow control information to a stream's source node, relies on the use of group communication, whereas in HammerHead, the use of group communication is restricted to inter-node communication within the cluster. Again, this decision was made to allow existing off-the-shelf media players to access a HammerHead service over a public network.

- In the existing framework, clients begin a new session by submitting a request to either the server group or one of the content groups. By enforcing a total order on these requests and on updates to the state of active sessions, the nodes can deterministically evaluate which node is the most suitable to handle the session. This thesis proposes an alternative model, described later in Chapter 5, in which initial client requests are distributed among cluster nodes and each client communicates with a single node, removing the need to totally order the delivery of messages for the implementation of load-balancing and thereby reducing the group communication overhead.

Vogels and van Renesse [Vv94] describe a development of the Berkeley Video-on-Demand service [RS92] using the Horus group communication toolkit [vBM96]. This system was based on the use of striping across a set of multimedia servers, as described in Chapter 2. The set of servers containing the segments of a stream requested by

a client form a process group and requests for stream segments are multicast to the group, hiding the structure of the multimedia server from clients. This work differs fundamentally from the work described in this thesis since it is based on the use of striping, rather than replication, and since it relies on a customized client media player that implements a proxy-at-client stream reconstruction scheme, as illustrated in Figure 2.12. In addition, this work uses a group communication toolkit for client-server communication, whereas the architecture described in this thesis only uses group communication internally within the server cluster.

A more comprehensive list of examples of the use of group communication, outside the domain of multimedia streaming services, appears in a survey by Chockler et al. [CKV01].

3.3 Summary

This chapter has described a number of existing dynamic replication policies. In a server cluster environment, both the *DASD Dancing* and *Dynamic Segment Replication* schemes would require active client streams to be migrated between cluster nodes and this thesis makes no assumption about either the clients' or servers' ability to perform such migration. Each invocation of the *Optimized Replication and Placement* scheme is independent and does not take into account the current assignment of replicas to nodes when re-evaluating the assignment, leading to excessive inter-node replication and reducing the bandwidth available for supplying client streams. *Threshold-Based Replication* – which creates additional replicas of a presentation when the bandwidth available for supplying streams of the presentation falls below some threshold – will tend to delay replication until service load is high. In contrast, this thesis proposes an approach in which replication is based on the relative demand for a presentation and is independent of the current service load. *Greedy Pseudo-Allocation* greedily assigns presentations to nodes, beginning with the most demanding presentation. Such a scheme

will create fewer replicas of the most demanding presentations whereas, intuitively, creating more replicas of the most demanding presentations will allow greater flexibility to perform load-balancing when servicing client requests. Finally, the *Bandwidth-to-Space Ratio (BSR)* policy attempts to match the BSR of storage devices with the cumulative BSR of the presentations stored on those devices. In contrast, this thesis proposes an approach that prioritizes load-balancing based on bandwidth over balancing of storage utilization.

The existing MMPacking [SGB98] replication algorithm, which has a number of attractive properties, has been adopted and significantly improved for use in the HammerHead multimedia server cluster architecture, which is described in Chapter 5. The proposed policy, called *Dynamic RePacking* assigns replicas of presentations to cluster nodes based on the proportion of the cluster bandwidth required to supply the mean number of concurrent streams of each presentation. As a result, the assignment of replicas to nodes is independent of actual service load. Each invocation of Dynamic RePacking takes into account the existing assignment of replicas to cluster nodes, to reduce the cost of adapting to changes in the relative popularity of presentations. Like MMPacking, Dynamic RePacking creates a near-minimal number of replicas to achieve load-balancing. A configurable minimum number of additional replicas of each presentation can be created to increase availability and these additional replicas are assigned to nodes in a manner that allows load-balancing to be maintained when nodes fail. Dynamic RePacking can assign presentations with different playback durations and bit-rates to nodes with different bandwidth and storage characteristics. Both MMPacking and the new Dynamic RePacking policy are described in detail in the next chapter.

Each of the dynamic replication policies described in this chapter was evaluated either through simulation or analytical studies and there does not appear to be an implementation of any of these policies described elsewhere in the literature. The performance of the Dynamic RePacking policy presented in the next chapter has been extensively evaluated using a prototype implementation of the HammerHead server

cluster architecture, as well as an event-driven simulation, and the results of these studies are presented in Chapter 6.

Although the use of group communication to implement on-demand multimedia streaming server clusters has been investigated in the past, the existing work differs from the HammerHead architecture in a number of important ways. Both of the existing architectures use customized media player applications to allow streaming client processes to participate in group communication, whereas HammerHead limits the use of group communication to inter-node communication within the cluster. This approach has the advantage of allowing unmodified commodity media players, which are in widespread use, to interact with a HammerHead server cluster. One of the existing architectures is based on striping, while the other, like HammerHead, is based on replication but assumes that presentations are statically assigned to nodes and does not address the implementation of a dynamic replication policy, as HammerHead does.

Chapter 4

Dynamic RePacking

On-demand multimedia server clusters based on dynamic replication of content assign a subset of the presentations in a multimedia archive to each cluster node. The subset assigned to each node will usually not be disjoint, with replicas of some presentations created on more than one node to facilitate load balancing, satisfy high client demand or increase the availability of individual presentations. For homogeneous systems, with cluster nodes with the same bandwidth and storage characteristics, supplying streams of presentations with the same bit-rates and durations, the probability of rejecting a request is minimized when each node has the same probability of being accessed [LV95]. A number of existing dynamic replication algorithms from the literature were described in Chapter 3.

This chapter describes the Dynamic RePacking algorithm, developed for the HammerHead multimedia server cluster architecture. The algorithm is a substantial development of the existing MMPacking algorithm proposed by Serpanos et al. [SGB98]. Chapter 2 described how the relative client demand for presentations and the structure of a server cluster can change over time and Dynamic RePacking has been designed to reduce the cost of adapting to such changes. In light of recent experience with the provision of large-scale highly-available services over the Internet, Dynamic RePacking has also been designed to separate replication for load-balancing from replication to increase availability. Dynamic RePacking creates a near-minimal number of replicas

to achieve load-balancing and allows a configurable minimum number of additional replicas of each presentation to be created to increase availability, providing a configurable trade-off between availability, performance and storage cost. In an extension of the policy, replicas of presentations are assigned to nodes in a manner that maintains load balancing when nodes fail. Unlike the original MMPacking algorithm, Dynamic RePacking can assign presentations with different playback durations and bit-rates to clusters of nodes with different bandwidth and storage characteristics.

4.1 Demand Estimation

It is assumed that the future demand for each presentation must be estimated by a HammerHead server cluster, before using the Dynamic RePacking policy to determine a suitable assignment of presentations to cluster nodes. The method used by the prototype HammerHead server cluster to estimate demand is described below. The resulting estimate of the demand for each presentation, together with a matrix describing the current assignment of presentations to nodes, forms the input to Dynamic RePacking, which is described later in the chapter.

When evaluating the demand for a presentation, several variables need to be considered:

Server load Changes in overall server utilization may occur on a short-term basis (e.g. between morning and evening). Usually we want to ignore these fluctuations and only respond to longer-term changes in the *relative* demand for presentations, for example, over several days.

Popularity The relative popularity of individual presentations will change over time, often decreasing as presentations become older and new presentations are introduced into the archive.

Stream duration The mean duration of streams of different presentations will usually vary, either because the streams themselves have different playback durations, or

because, on average, clients only view part of each presentation. An example of this latter case might be an archive of news broadcasts, where a significant number of clients are only interested in the news headlines. To see why this is important, consider two presentations, A and B, each of which receives twenty requests per hour. If the mean duration of streams of A is twice that of B then, according to Little's formula [Lit61], there will be twice as many concurrent streams of A relative to B, as illustrated in Figure 2.6.

Bit-rate The presentations stored in an archive will usually be encoded with different bit-rates, depending, for example, on the media type (video or audio) or the level of compression used. Again, the impact of such differences is illustrated in Figure 2.6.

The mean number of concurrent streams, L_i, of each presentation i over a period of length τ can be determined by applying Little's formula [Lit61] as follows:

$$L_i = \lambda_i W_i \tag{4.1}$$

The arrival rate, λ_i can be calculated by dividing the number of requests for the presentation, R_i, by the length of the period, τ. Similarly, the mean stream duration, W_i, can be calculated by dividing the cumulative duration of all streams of the presentation during the period by the number of requests, giving:

$$L_i = \frac{R_i}{\tau} \cdot \frac{Q_i}{R_i} = \frac{Q_i}{\tau} \tag{4.2}$$

This value can be scaled by the bit-rate, B_i, of the presentation to give an expression for the server bandwidth required to supply the mean number of concurrent streams of the presentation during the evaluation period.

However, as stated above, short-term changes in overall server utilization, such as those that occur between on-peak and off-peak times, should be ignored and Dynamic RePacking should only respond to changes in the *relative* demand for each presentation.

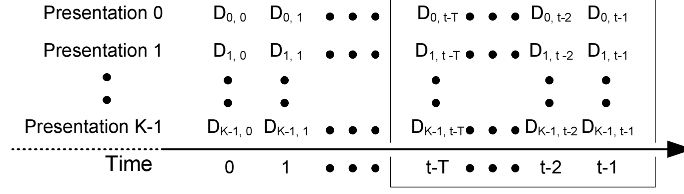


Figure 4.1: Saved measurements of demand used to estimate $D'_{i,t}$

For this reason, the demand, D_i, of a presentation i is defined as the *proportion* of server bandwidth required to supply the mean number of concurrent streams of the presentation, which is independent of the actual load on the server. Thus, D_i can be evaluated for each presentation using the following formula:

$$D_i = \frac{Q_i B_i}{\sum_{k=0}^{K-1} (Q_k B_k)} \tag{4.3}$$

where K is the total number of presentations in the archive. Thus, to estimate the demand for each presentation, we only need to measure the cumulative duration, Q_i, of each presentation over the period τ.

To obtain a mean value for D_i over a longer period, the last T values for D_i are saved, as shown in Figure 4.1, and are used to calculate a weighted average demand. The estimated demand, $D'_{i,t}$, for a presentation at time t is evaluated as follows:

$$D'_{i,t} = \frac{\sum_{x=1}^{T} (D_{i,t-x} w_x)}{\sum_{x=1}^{T} w_x} \tag{4.4}$$

where $D_{i,x}$ is the demand for presentation i during period x. Multiplying each term by w_x can be used to give more weight to more recent measurements.

Alternatively, an exponential average scheme that allows the weight given to past measurements of demand to be controlled might be used, with the estimated demand $D'_{i,t}$ given by:

$$D'_{i,t} = \alpha D_{i,t-1} + (1 - \alpha) D'_{i,t-1} \tag{4.5}$$

If $\alpha = 1$, then the estimated demand is based only on the demand measured during

the last period of length τ. Otherwise, as α increases towards 1, more weight is given to past measurements.

At the expense of additional computation time, schemes based on the extrapolation of recorded measurements of $D_{i,t}$ could also be used.

4.2 Replica Assignment

Once the relative demand for each presentation has been estimated, replicas of each presentation need to be assigned to cluster nodes such that load-balancing (based on expected demand) is achieved and any additional requirements, for example, the creation of additional replicas to increase availability, are satisfied. The *Dynamic RePacking* algorithm was developed for the prototype HammerHead server cluster and is based on the MMPacking algorithm proposed by Serpanos et al. [SGB98]. The MMPacking algorithm and its advantages and disadvantages are described below.

4.2.1 MMPacking

Consider a multimedia server cluster with N nodes, $S_0 \ldots S_{N-1}$, that stores K multimedia presentations, $M_0 \ldots M_{K-1}$. Each presentation M_i has popularity p_i. It is assumed that $K \gg N$ and that each node has the same bandwidth. Initially the presentations are sorted in ascending order of popularity, such that $p_0 \leq p_1 \leq \ldots \leq p_{K-1}$.

The MMPacking algorithm proceeds as follows. Replicas are assigned to nodes in a round-robin fashion, beginning with the least popular presentation and proceeding in order of increasing popularity, as illustrated in Figure 4.2(a). The popularity of a node is the aggregate popularity of the replicas assigned to that node. If, after assigning a replica to a node, the aggregate popularity of the node exceeds $1/N$, then the popularity of the replica being assigned to the node is reduced such that the aggregate popularity of the node will be equal to $1/N$. Another replica of the same presentation, with the excess unallocated popularity, is assigned to the next node in round-robin order (Figure 4.2(b)). Finally, if after assigning a second or subsequent replica of a

presentation to a node, the node's aggregate popularity is less than $1/N$, then the algorithm continues by assigning the first replica of the next presentation to the *same* node (Figure 4.2(b)), thereby achieving an aggregate popularity of $1/N$ for the node. A final assignment of replicas to nodes produced by the MMPacking algorithm is illustrated in Figure 4.2(c). When the popularity of each presentation has been satisfied, the aggregate popularity of each node will be exactly $1/N$. Pseudo-code for the MMPacking algorithm is shown in Figure 4.3. The `AssignReplica(...)` function assigns a replica to a node and the `NextNode(...)` function selects the next node as follows:

```
ndIdx = ( ndIdx + 1 ) MOD N
```

The MMPacking algorithm has a number of attractive properties.

- The aggregate popularity of the replicas assigned to a node will be exactly the same for each node. In other words, if the estimated popularity of the presentations in the archive is exactly equal to their actual popularity, then optimal load balancing can be achieved.
- Because replicas are assigned to nodes in order of increasing presentation popularity, the algorithm results in the assignment of replicas of both popular and unpopular presentations to each node, as recommended by Dan and Sitaram [DS95b].
- As well as balancing the expected aggregate popularity of each node, MMPacking also achieves storage balancing. Specifically, it was proven that there is a difference of at most two between the number of replicas assigned to any pair of nodes. It can also be shown that each node will be assigned at most $K/N + 1$ replicas. Since the minimum per-node storage capacity required to store exactly one replica of each presentation on the cluster is K/N, the storage overhead, due to the creation of additional replicas by MMPacking, is small.
- If MMPacking creates more than one replica of a presentation, all presentations with higher popularity will also have more than one replica. Since MMPacking

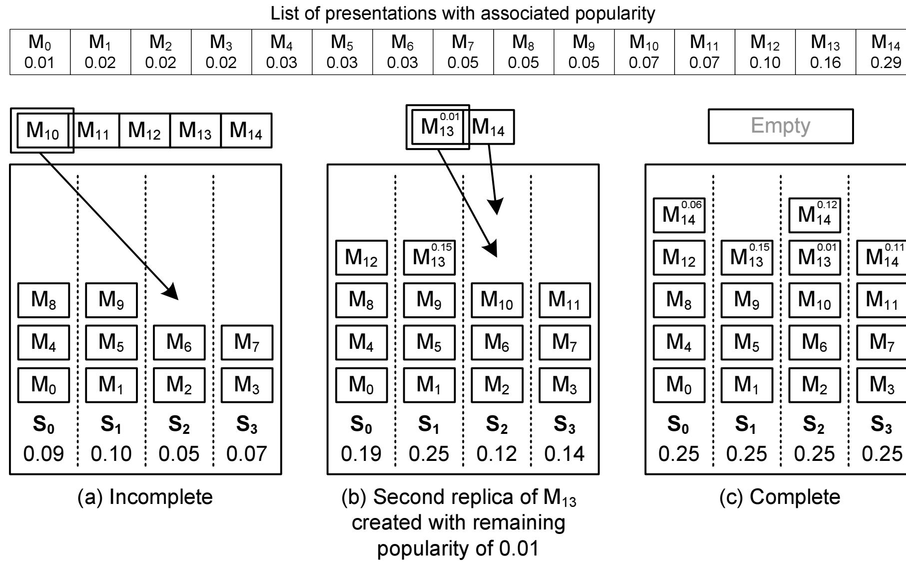


Figure 4.2: Illustration of the MMPacking algorithm, showing the popularity assigned to presentations with more than one replica and the popularity of each node

```
presentations.SortUp();
presIdx = 0;
while(presentations.count > 0) {
  allocation = MIN(1/N - nodes[ndIdx].popularity, presentations[presIdx].popularity);
  AssignReplica(nodes[ndIdx], presentations[presIdx]);
  nodes[ndIdx].popularity += allocation;
  presentations[presIdx].popularity -= allocation;
  presentations[presIdx].replicaCount++;

  if(presentations[presIdx].replicaCount == 1 || nodes[ndIdx].popularity == 1/N)
    ndIdx = NextNode(ndIdx);

  if(presentations[presIdx].popularity == 0)
    presentations.Remove(presIdx)
}
```

Figure 4.3: MMPacking algorithm

results in the creation of at most $N - 1$ additional replicas to achieve load balancing (beyond the minimum of one replica for each presentation), then only a small number of the most popular presentations will have more than one replica. Intuitively, this means that the presentations for which most requests are received provide the greatest flexibility to select a node to service the request and thereby achieve load-balancing.

The MMPacking algorithm, however, also has a number of disadvantages.

- While MMPacking achieves storage balancing in terms of the number of replicas assigned to each node, unless all of the replicas are of uniform size, the storage capacity required by each node may vary. One might, however, offer the counter-argument that since the number of presentations is significantly larger than the number of nodes, the average size of the replicas stored on each node will be approximately the same.

- The MMPacking algorithm assumes the storage capacity of each node is unconstrained. In practice, there may be insufficient storage capacity on a node to assign a replica of the next presentation, preventing the algorithm from proceeding.

- MMPacking is unable to handle a server cluster containing heterogeneous nodes with non-uniform bandwidth capacities, since the aggregate popularity of the replicas assigned to a node will be $1/N$, regardless of the bandwidth of the node.

- Assigning replicas to nodes according to the *popularity* of presentations, rather than the proportion of available service capacity required to supply the average number of concurrent streams of each presentation (which was discussed earlier and is referred to as *demand*), will not result in load-balancing between cluster nodes. For example, consider two presentations, M_i and M_j, each with the same popularity but with streams of the second lasting twice as long as streams of the first. The proportion of server resources required by the second presentation

will be twice that required by the first. The same problem is encountered when packing presentations with different bit-rates.

- Each invocation of the algorithm is independent of any previous invocation. As a result, even a minor change in the ranking of presentations by popularity will frequently result in the majority of the replicas being moved between nodes. Since it is desirable to frequently reevaluate the estimated popularity of each presentation and the assignment of replicas to nodes, MMPacking would result in the use of a significant proportion of server resources for the creation and movement of replicas, rather than supplying streams to clients.

4.2.2 Dynamic RePacking

Since the assignment of replicas to nodes is based only on an estimate of the relative demand for each presentation, Dynamic RePacking – like the other dynamic replication policies that have been proposed in the past and described in Chapter 3 – uses a heuristic approach, as suggested by Dan and Sitaram [DS95b], rather than producing an optimal distribution of the replicas, which would minimize the storage capacity required.

Dynamic RePacking bases its allocation of replicas to nodes on the *demand* for each presentation – the representation and estimation of which was described earlier. The aggregate demand for all of the presentations stored on a cluster is equal to one. Dynamic RePacking aims to assign replicas to nodes such that the aggregate demand for the replicas on each node is equal to the *target aggregate demand* for the node. If each node in a cluster has the same service capacity, then the target aggregate demand for each node will be $1/N$, where N is the number of nodes in the cluster.

Unlike MMPacking, however, Dynamic RePacking does not assume that each node has the same service capacity and hence, the same target aggregate demand. Instead, the target aggregate demand of a node j is determined by expressing the service capacity of the node (which, it is assumed, is expressed in terms of the bandwidth, B_j, available for supplying multimedia streams) as a proportion of the service capacity of

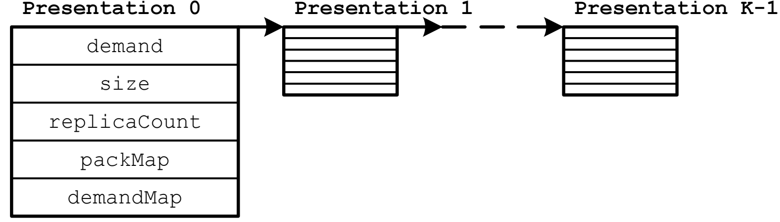


Figure 4.4: List of `Presentation` objects used by Dynamic RePacking

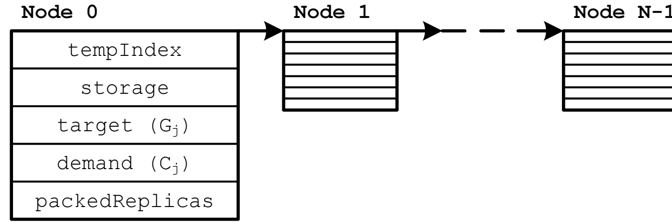


Figure 4.5: List of `Node` objects used by Dynamic RePacking

the whole cluster. Formally, the target aggregate demand, G_j for a node j is defined as follows:

$$G_j = \frac{B_j}{\sum_{n=0}^{N-1} B_n} \tag{4.6}$$

As the Dynamic RePacking algorithm proceeds and replicas of presentations are assigned to nodes, the aggregate demand for the replicas assigned to each node will change. The *target shortfall*, H_j, of each node is defined as the difference between the aggregate demand for the replicas assigned to the node so far and the target aggregate demand, G_j, of the node. Formally, if the aggregate demand for the replicas assigned to a node j is C_j, then the target shortfall, H_j, is:

$$H_j = G_j - C_j \tag{4.7}$$

Figures 4.4 and 4.5 illustrate the data structures created and maintained by the Dynamic RePacking algorithm during the assignment of replicas to nodes. The presentation list is a list of `Presentation` objects, which maintain information about the

presentations stored on the cluster. The estimated demand for a presentation is represented by `demand` and the physical size of the presentation (its file size) is represented by `size`. An array of integers, `packMap`, with an element for each cluster node, stores the current and new location of the replicas of a presentation and is explained in detail later. As Dynamic RePacking creates replicas of a presentation, the corresponding `Presentation` object's `replicaCount` is incremented. Finally, `demandMap` is an array of floating-point values, with one element for each cluster node, used to record how the demand for each presentation is distributed among its replicas.

The node list is a list of `Node` objects, which maintain information about the cluster nodes to which replicas will be assigned. A temporary index, `tempIndex`, is assigned to each `Node` object before executing the Dynamic RePacking algorithm and is used to index the `packMap` and `demandMap` arrays of the `Presentation` objects. The target aggregate demand, G_j, of a node is represented by `target`. When Dynamic RePacking assigns a replica to a node, the remaining storage capacity – represented by `storage` – is reduced by the file size of the replica, the aggregate demand for the node – represented by `demand` – is increased by the replica's demand and the corresponding `Presentation` object is added to the `packedReplicas` list of replicas assigned to the node.

The `packMap` member of a `Presentation` fulfills the same role as the assignment matrix described earlier in Chapter 3. Instead of maintaining "before" and "after" assignment matrices, however, the `packMap`s are equivalent to a single "difference" matrix, which is modified as the Dynamic RePacking algorithm proceeds. The `packMap` arrays of each `Presentation` object may be viewed as the columns of the matrix.

The `packMap` associated with each `Presentation` object is an array of integers with N elements, where N is the number of nodes in the cluster. The ith element of the array corresponds to the `Node` object with `tempIndex` i and initially contains a value between -2 and 0. Before Dynamic RePacking begins, the elements of the `packMap` are initialized as indicated by Table 4.1. When Dynamic RePacking dictates that a replica of a presentation should be assigned to a node, the `packMap` element corresponding to

Value	Meaning
-2	Node j does not contain a replica of the presentation
-1	Node j contains a replica that is scheduled for deletion
0	Node j contains a replica (*not* scheduled for deletion)

Table 4.1: Meaning of `packMap` values before Dynamic RePacking

Value	Meaning
-2	Node j still does not contain a replica (no change)
-1	Node j still contains a replica that is scheduled for deletion (no change)
0	Node j contains a replica which should now be deleted
$+1$	A new replica should be created on node j
$+2$	The replica scheduled for deletion on node j should be undeleted
$+3$	Node j has a replica which should remain (no change)

Table 4.2: Meaning of `packMap` values after Dynamic RePacking

the node is incremented by 3. After Dynamic RePacking has completed, each `packMap` element contains a value between -2 and $+3$, with the meaning of each value shown in Table 4.2. The value of a `packMap` element indicates what action, if any, needs to be taken to achieve the assignment of replicas determined by the Dynamic RePacking algorithm. For example, a value of 0 would indicate that an existing replica should be removed, while a value of $+2$ indicates that a replica scheduled for deletion should be undeleted.

Figure 4.6 illustrates the state of the `Node` and `Presentation` objects for the Dynamic RePacking example shown later in Figure 4.9. (Specifically, this example illustrates the state of the data structures corresponding to Figure 4.9 (d).)

Before Dynamic RePacking begins, the `replicaCount` of each `Presentation`, and every element of its `demandMap`, will be zero. Each `packMap` is initialized as indicated in Table 4.1. Each `Node` object's `demand` is zero, its `packedReplicas` list is empty and its `storage` is the available storage capacity before assigning any replicas to the node. Dynamic RePacking proceeds from this initial state by attempting to assign the replicas of each presentation to nodes that already contain a replica of the same presentation

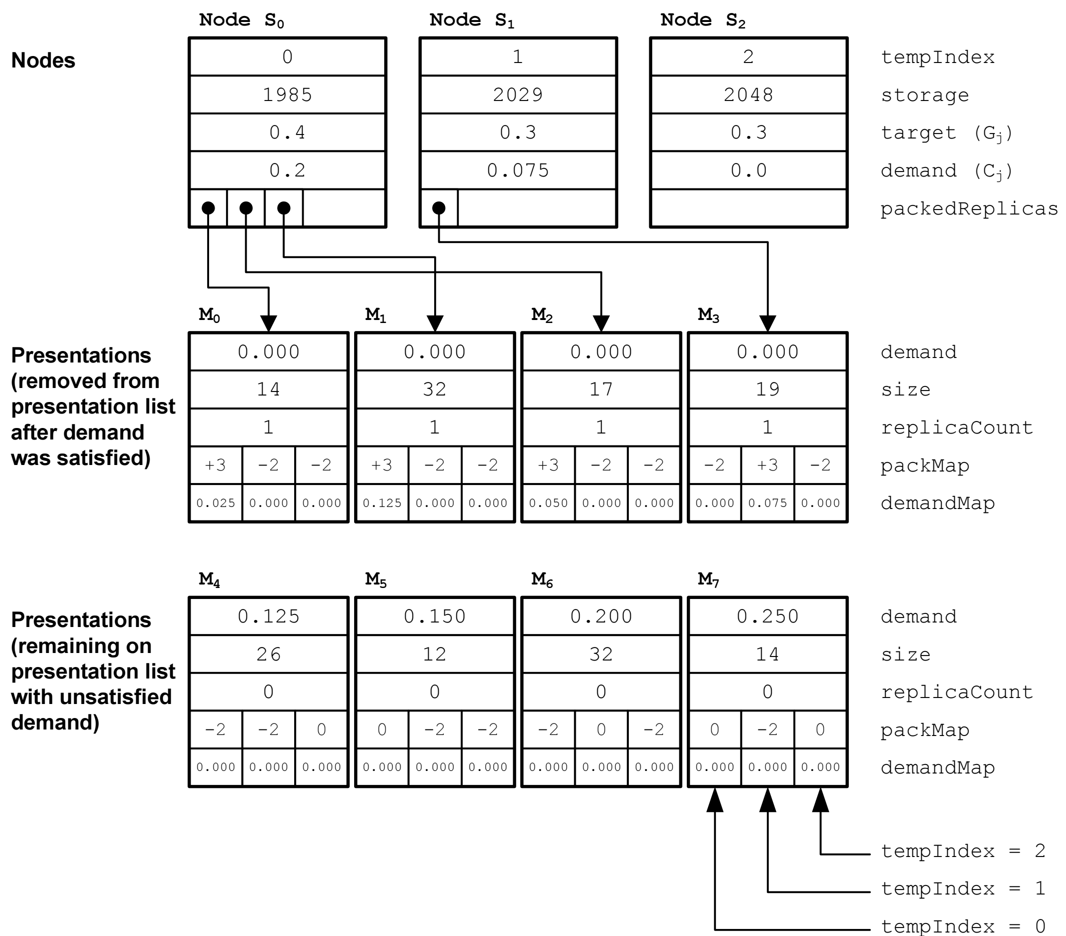


Figure 4.6: Illustration of the use of `Node` and `Presentation` objects

from a previous invocation of the algorithm, using the information contained in the packMap.

Like MMPacking, Dynamic RePacking assigns replicas to nodes in round-robin order and the algorithm may be viewed as consisting of a number of rounds. The list of Node objects, illustrated in Figure 4.5, is sorted by decreasing target shortfall, H_j^l, at the beginning of each round $l + 1$ of assignments. The list of Presentation objects is sorted by increasing estimated demand. Each round of assignments begins with the first Node object in the node list. The presentation whose replica is selected for assignment to the current node is that represented by:

- the first Presentation object in the presentation list *that already has a replica on the current node, from a previous invocation of Dynamic RePacking* (packMap value is 0 or -1)

  or, if no such Presentation exists

- the first Presentation object in the list, *regardless of whether a replica is already stored on the node.*

The motivation for selecting replicas for assignment in this way is to reduce the cost of adapting to changes in the relative demand for individual presentations. As described in section 4.2.1, even a minor change in the ranking of presentations would be likely to cause MMPacking to relocate a significant number of replicas.

The actions performed to update the Node and Presentation objects when assigning a replica to a node are summarized by the pseudo-code in Figure 4.7. By assigning a replica to a node during round l, the target shortfall, H_j^l, is reduced by the demand associated with the replica's presentation. If assigning a presentation to a node completely satisfies its demand, the associated Presentation object is removed from the presentation list. If, however, the demand for the presentation exceeds the remaining target shortfall of the node, the replica is assigned to the node, the demand of the Presentation object is reduced by the target shortfall of the node and the object remains on the presentation list for later assignment to another node, thus creating

```
PackReplica(node, presentation)
{
  allocation = MIN(presentation.demand, node.target - node.demand);
  presentation.packMap[node.tempIndex] += 3;
  presentation.demandMap[node.tempIndex] = allocation;
  presentation.demand -= allocation;
  presentation.replicaCount++;

  node.demand += allocation;
  node.storage -= presentation.size;
  node.packedReplicas.Add(presentation);
}
```

Figure 4.7: Pseudo-code for the `PackReplica()` function

an additional replica of the presentation. In this latter case, the presentation list is re-sorted in increasing order of demand.

Each round of assignments ends when the target shortfall of the current node, after assigning a replica, is still greater than the target shortfall of the next node, or when the end of the list of `Node` objects is reached. Formally, a round of assignments ends if either of the following conditions is true:

$$H_j^l > H_{j+1}^{l-1}, \qquad \text{for } 0 \leq j \leq N-2 \tag{4.8}$$

or

$$j = N - 1 \tag{4.9}$$

The former condition would never be satisfied when using MMPacking but may be satisfied when using Dynamic RePacking either because the cluster nodes have heterogeneous bandwidth capacities or because the assignment of replicas to a node favours those presentations with replicas already stored on the node. Ending rounds of assignment early under these conditions prevents the creation of additional replicas as a result of the assignment of replicas to those nodes with a lower target aggregate demand, while the same replicas could be assigned elsewhere, without creating additional replicas. The creation of additional replicas is thus more likely to be deferred until only the most demanding presentations remain to be assigned, which, it was argued previously, was one of the attractive properties of the original MMPacking algorithm.

```
presentations.SortUp();
ndIdx = 0;

while(presentations.count > 0) {
  if(ndIdx == 0)
    nodes.SortDown();

  if(nodes[ndIdx].demand < nodes[ndIdx].target) {
    presIdx = SelectObject(ndIdx);
    PackReplica(nodes[ndIdx], presentations[presIdx]);

    if(presentations[presIdx].replicaCount == 1 || nodes[ndIdx].demand == nodes[ndIdx].target)
      ndIdx = NextNode(ndIdx);

    if(presentations[presIdx].demand == 0)
      presentations.Remove(presIdx);
    else
      presentations.SortUp();
  } else {
    ndIdx = NextNode(ndIdx);
  }
}
```

Figure 4.8: Pseudo-code for the Dynamic RePacking algorithm

Pseudo code for the basic Dynamic RePacking algorithm is shown in Figure 4.8 and an assignment of replicas to nodes is illustrated in Figure 4.9. Figure 4.9 (a) shows an existing assignment of replicas to nodes, before executing the Dynamic RePacking algorithm. Packing begins with node S_0 and continues with this node until its target shortfall is less than the target shortfall of the next node, S_1 (Figures 4.9 (b), (c) and (d)), after which packing continues as described. After repacking M_4, the round of assignments ends, the list of nodes is reordered by decreasing target shortfall and repacking continues with M_6 on S_1 (Figures 4.9 (e) and (f)). Repacking M_7 on S_2 reduces the target shortfall of S_2 to zero and presentation M_7 remains on the presentation list (Figure 4.9 (g)). The final pack is shown in Figure 4.9 (h). The result of the pack was the movement of one of the replicas of M_7 from S_0 to S_1.

4.2.3 Constrained Storage Capacity

When assigning a replica of a presentation to a node, during the execution of the Dynamic RePacking algorithm, there may be insufficient storage capacity remaining on the node to store the replica. In this case, replicas of the least demanding presentations, already assigned to the node by Dynamic RePacking during previous rounds,

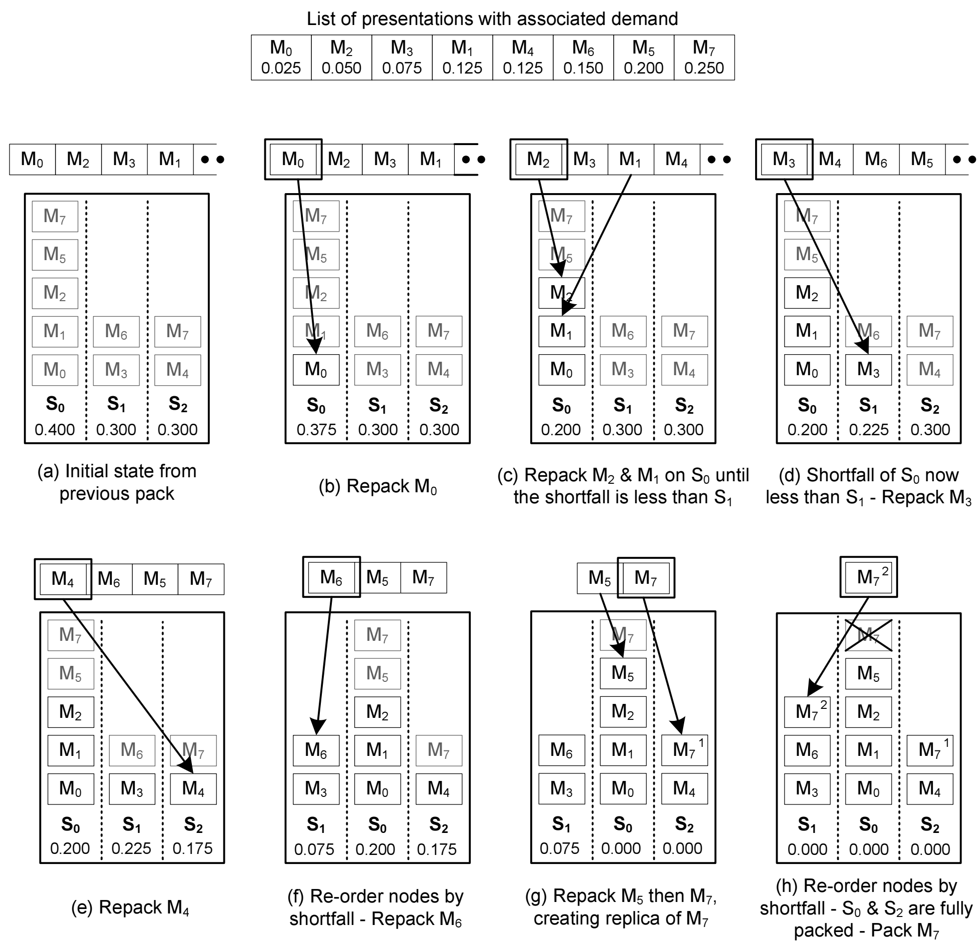


Figure 4.9: Illustration of the Dynamic RePacking algorithm showing the target shortfall of each node

are unpacked in increasing order of demand, until there is sufficient storage capacity to pack the next replica. Any unpacked presentations are placed on a victim list. If unpacking replicas in this way, to allow the next replica to be packed, would result in an *increase* in the target shortfall for the node, packing the current replica is aborted and the algorithm continues by attempting to pack a replica of the next most demanding presentation that meets the selection criteria defined earlier. In other words, assigning a replica to a node should never result in a reduction in the aggregate demand for the replicas assigned to the node and should always progress the algorithm towards achieving a target shortfall of zero for each node. If no replica can be packed in this way, the node is removed from the node list and no further attempt is made to assign replicas to it.

After Dynamic RePacking has completed, replicas of those presentations on the victim list are packed on nodes with sufficient storage capacity. Since the replicas on the victim list will have low expected demand, the impact on load-balancing will be small. In order of decreasing demand, a single replica of each presentation on the victim list is placed on the node with the most available storage capacity, favouring those nodes that already have a replica of the presentation from a previous invocation of the Dynamic RePacking algorithm. If a presentation cannot be assigned to a node, the algorithm moves on to attempt to pack a replica of the next presentation on the victim list and the unpacked victim presentation will not be stored on the cluster.

4.2.4 Fault-Tolerant Dynamic RePacking

The basic Dynamic RePacking algorithm assigns a replica of each presentation to just a single node, unless the creation of additional replicas is required to perform load-balancing. With just one replica of each node, however, the failure of any node will result in a reduction in harvest. By creating additional replicas, the reduction in harvest due to node failures can be reduced. For example, if every presentation was assigned to at least two nodes, harvest would be maintained at 100% for single node failures and would only be reduced if a second node failed before replacement replicas had been

created. Dynamic RePacking provides a mechanism through which additional replicas can be created and packed on cluster nodes in a way that maintains load-balancing when nodes fail.

Harvest does not take the relative value of individual presentations into account. For example, the loss of a popular presentation would have a significant impact on the yield of the service, whereas the loss of a relatively unpopular presentation might have little or no effect. The scheme described here allows a minimum number of replicas to be specified for some proportion of the most demanding presentations. For example, a service might require that the most demanding 10% of the presentations should be packed on at least two nodes. Given a minimum number of replicas to be created for a subset of the presentations (`minPack`) and a floating point value between 0 and 1 specifying the proportion of the most demanding presentations to be included in this subset and to which the `minPack` minimum should be applied, the `minReplicaCount` attribute of each `Presentation` in this subset is set to `minPack`, with the remaining `Presentation` objects having a `minReplicaCount` of one.

Alternative schemes may be used to determine the minimum number of replicas to be created for each presentation. For example, additional replicas might be created for the most *popular* 10% of the presentations, rather than the most *demanding* 10%. Alternatively, a scheme such as Webster's monotone divisor apportionment method could be used to assign additional replicas to presentations in proportion to their demand, which is the approach used by the DASD Dancing replication scheme [WYS95]. In practice, the choice of a suitable scheme to determine the minimum number of replicas required for each presentation will be application specific, and fault-tolerant Dynamic RePacking has been designed to be independent of the actual scheme used.

After the `minReplicaCount` attribute of each `Presentation` object has been set, the basic Dynamic RePacking algorithm described earlier is used iteratively, in phases, to assign replicas to nodes such that load-balancing is achieved, load-balancing can be maintained in the presence of node failures and, with sufficient storage capacity, each presentation is assigned to at least the number of nodes specified by the

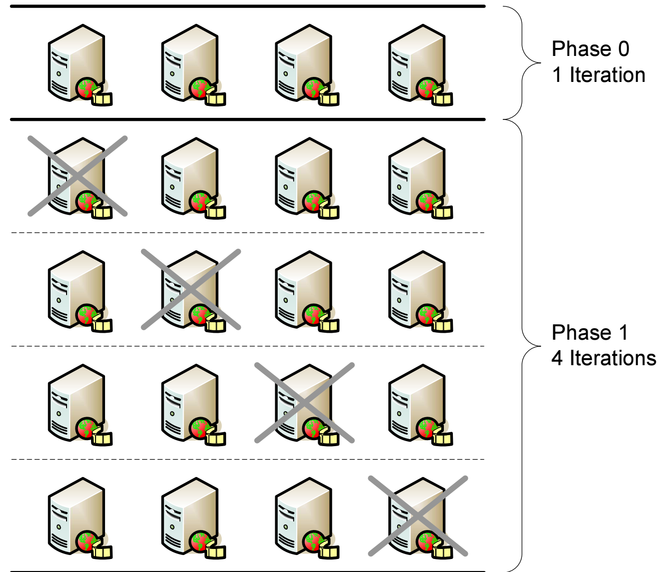


Figure 4.10: Dynamic RePacking over two phases for a four-node cluster

`minReplicaCount` attribute of the corresponding `Presentation` object.

In each phase f, the presentations are packed with f nodes excluded, as if those nodes had failed. Dynamic RePacking is executed $\binom{N}{f}$ times in each phase f, and for each execution, the presentations are packed as though a different combination of f nodes had failed. Thus, for $f = 0$, the presentations are packed with no failed nodes and Dynamic RePacking is executed $\binom{N}{0} = 1$ times, corresponding to the basic Dynamic RePacking algorithm described in the previous section.

For each subsequent phase, with $f > 0$, Dynamic RePacking is executed for each combination of f "failed" and $N - f$ non-failed nodes, with the failed nodes excluded, as shown in Figure 4.10. In addition, only those presentations that have either not yet achieved the required minimum number of replicas (`minReplicaCount`), or have a replica remaining on a non-failed node, are packed. In other words, those presentations that have reached their required minimum number of replicas, all of which exist only on the excluded nodes, will not be packed by Dynamic RePacking in the current iteration.

When performing fault-tolerant Dynamic RePacking in this way, the criteria for selecting a presentation to be assigned to a node is modified such that the selected presentation is that represented by:

- the first `Presentation` in the presentation list which, if already packed `minReplicaCount` times, was packed on the current node *during this reevaluation of the assignment of replicas to nodes* (`packMap` value is greater than 0) or, if not yet packed `minReplicaCount` times, exists on the node *from a previous evaluation of the assignment of replicas to nodes* (`packMap` value is 0 or -1).

  or, if no such presentation exists

- the first presentation in the presentation list that exists on the node *from a previous evaluation of the assignment of replicas to nodes*, regardless of whether it has been packed `minReplicaCount` times.

  or, if no such presentation exists

- the first presentation in the presentation list, *regardless of whether it exists on the current node from a previous evaluation of the assignment of replicas to nodes.*

The last two selection criteria are the same as those for the basic Dynamic RePacking algorithm. The motivation for the first criterion is to avoid creating additional replicas of those presentations that already have `minReplicaCount` replicas, while creating additional replicas of those presentations that still have not reached their `minReplicaCount`. Both of these sets of presentations will be given equal priority and selected in increasing order of demand. By interleaving the selection of both of these sets of replicas, the additional replicas originally stored on each "failed" node will be distributed among the remaining nodes. As a result, the workload associated with those presentations whose replicas were assigned to a node that actually failed, would be distributed across the remaining nodes.

The number of packing phases performed must be at least equal to the value of `minPack` used to specify the desired minimum number of replicas of each presentation. In other words, for example, if there are `Presentation` objects with a `minReplicaCount` of two, then at least two packing phases are required. Large numbers of nodes and values of f greater than one will significantly increase the execution time of the fault-tolerant Dynamic RePacking algorithm. Since, however, the probability of a second node failing

before a first failure can be repaired will typically be small, it is expected that additional phases with f greater than one would be unnecessary. If further replication was required, a scheme that does not attempt to maintain load-balancing in the presence of node failures, such as that described by Wei and Venkatasubramanian [WV01], may be used to further increase the number of replicas of each presentation.

4.2.5 Fit Threshold

It will be shown in Chapter 6 that, under certain circumstances, such as may occur when assigning replicas to large numbers of nodes or nodes with heterogeneous performance and storage characteristics, Dynamic RePacking may result in the movement of a small proportion of replicas for little actual gain. To counteract this effect, a scheme is proposed to evaluate how well an existing distribution of replicas satisfies the new expected demand. The scheme simply applies the original (non-fault-tolerant) Dynamic RePacking algorithm described in section 4.2.2, but restricts the selection of a presentation to assign to each node to those presentations that have a replica on the node from a previous invocation of Dynamic RePacking (and, in addition, if using fault-tolerant Dynamic RePacking, only those replicas packed during phase 0 are considered). In other words, Dynamic RePacking is applied without the ability to create new replicas. On completion, those presentations whose demand has not been satisfied will remain on the presentation list. If the sum of the demand for these presentations is below a configurable value, referred to as the *Fit Threshold*, the current assignment of replicas is deemed to be adequate, otherwise, Dynamic RePacking is performed to reevaluate the assignment. It will be shown in Chapter 6 that applying this modification has the effect of preventing replication for little gain, while still allowing a cluster to adapt to changes in the relative demand for presentations.

4.3 Simulation

An event-driven simulation was used during the development of the Dynamic RePacking algorithm, to evaluate its performance, before implementing the algorithm in the prototype HammerHead server cluster. This simulation has been used to predict the performance of the HammerHead architecture for a wider range of cluster configurations and client workloads than was possible using the prototype cluster implementation, and the results of the simulation are presented in Chapter 6.

4.4 Summary

This chapter has described the Dynamic RePacking content replication policy, which improves on the existing MMPacking policy, allowing it to be applied to clusters with varying bandwidth and storage capacities and to multimedia archives containing presentations with varying bit-rates and durations. The algorithm significantly reduces the cost of adapting to changes in the relative demand for presentations. A scheme for estimating the proportion of a cluster's service capacity required to supply the average number of concurrent streams of each presentation has also been described and this scheme will be used to provide the input to the Dynamic RePacking algorithm in the prototype HammerHead server cluster, which is described in the next chapter.

A number of enhancements to the basic Dynamic RePacking algorithm have also been described, allowing the algorithm to proceed when the storage capacity of a cluster node is constrained and allowing additional replicas to be created to increase the availability of an on-demand multimedia streaming cluster. Fault-tolerant Dynamic RePacking assigns replicas to nodes in a manner that allows load-balancing to be maintained when nodes are removed from a cluster.

Chapter 5

The HammerHead Server Cluster

The HammerHead architecture is based on the use of selective replication of multimedia content in a cluster of commodity PC nodes. Each incoming client request needs to be redirected to one of the cluster nodes that has a replica of the requested presentation and, when there is more than one such node, the requests must be redirected to the most suitable node, to achieve load-balancing. The role of the Dynamic RePacking policy described in the previous chapter is to assign replicas to appropriate cluster nodes in a manner that allows load-balancing to be achieved, based on the estimated demand for each presentation. Redirecting initial client requests, estimating the demand for each presentation, implementing the Dynamic RePacking policy and managing the creation, movement and removal of replicas within the cluster all require the availability of information describing the current location and state of replicas and the current activity of each node, as well as historical information describing the past demand for each multimedia presentation.

The HammerHead architecture adds a cluster-aware layer to an existing stand-alone commodity multimedia server. The state of the local commodity server instance on each cluster node is combined to form an aggregated cluster state, which is replicated on some or all of the cluster's nodes. This state information is used to redirect client requests to multimedia server instances on suitable cluster nodes and to implement the Dynamic RePacking content replication policy described in Chapter 4. As the

state of the commodity multimedia server instance on each node changes over time, for example, when new streams begin or replicas are removed, multicast events are issued and each replica of the aggregated cluster state is updated accordingly. By replicating the aggregated cluster state, the client redirection task can be shared among multiple nodes and the implementation of the dynamic replication policy becomes tolerant to multiple node failures.

This chapter describes the HammerHead architecture in detail. The architecture is presented in the context of a prototype implementation, which relies on three key technologies – the Ensemble group communication toolkit, the Windows Media Services multimedia streaming server and the Microsoft Windows Network Load Balancing cluster service – and the chapter begins with a description of each of these technologies.

5.1 Group Communication using Ensemble and Maestro

Group communication describes a form of inter-process communication in which a group of distributed processes receive a common stream of messages. Messages are addressed to the group, rather than to its individual members. The messages sent in a group communication environment are *multicast* to the member processes. In contrast, messages addressed and sent to single processes are referred to as *unicast* messages, while *broadcast* messages are addressed to every process visible to the sender.

Like unicast communication systems, it may be necessary to impose certain ordering guarantees on the delivery of multicast messages in a group communication system [Bir96] and different applications will require different ordering guarantees. To discuss the ordering guarantees provided by a communication system, it is necessary to distinguish between the receipt of a message and its subsequent delivery. A message is *received* when it arrives at the host on which the process resides and messages may be received out-of-order or received more than once. A message is *delivered* when the application's ordering requirements have been satisfied and it is presented to the

process, which handles delivered messages in first-come-first-served order. For example, for certain applications, it may be sufficient for messages to be delivered in the same order in which they are received, without any further ordering. *FIFO* (First-In-First-Out) ordering delivers messages sent by the same process in the order in which they were originally sent, with no guarantees about the relative ordering of messages sent by different processes. *Causal* ordering delivers messages according to Lamport's happened-before relation – messages are delivered only after any messages which potentially affected the message have first been delivered. Other applications may require a stricter form of ordering, in which messages are delivered in some *agreed* or *total* order such that, if any process delivers any pair of messages in some order, the same pair of messages will be delivered in the same order at every process.

A group communication system requires a group membership service to manage the addition or removal of processes to or from the group, notify members of changes in the group's membership, detect the failure of group members and define the set of processes to which each multicast message is sent. Not all of these facilities will be present in the group membership service of a group communication system. For example, IP multicast, which can be considered a form of group communication [CDK01], does not provide failure detection or inform group members of changes in the membership of the group. The membership of a group is described by a *view* – an ordered set of identifiers of the processes currently in the group. Process identifiers in a view appear in the same order at every member process.

Processes can join groups, leave groups of which they are a member, or be excluded from a group if they are suspected of failing by the group membership service. A new group is created when a process joins a previously empty group and groups may subdivide or merge together, when network partitions or reconnections occur. When the membership of a group changes, the group membership service *delivers* a new view to each member of the group.

5.1.1 View-Synchronous Group Communication

Group communication systems such as Ensemble, which is described later, can provide certain guarantees about the ordering of view delivery with respect to other inter-process messages, producing a *view-synchronous* group communication system [Bir96, CDK01]. Such systems deliver the same sequence of views in the same order to each group member and each process delivers the same set of messages between any two consecutive views. If a sender fails to deliver a message to a group member, a new view that excludes the failed member is delivered to the remaining processes, immediately after the view in which the remaining members delivered the message. If the sender of a message fails, then either no process delivers the message or all surviving processes deliver it. In the latter case, the surviving processes deliver the message in a view that contains the original sender, before the new view excluding the sender is delivered.

Figure 5.1 (a) illustrates two multicast messages sent by processes p_0 and p_1. The first message is sent before process p_3 joins the group and is not delivered to that process. The second message is sent as the new view is being installed and is delivered in that new view to processes $p_0 \ldots p_3$. The same scenario with incorrect behaviour for both messages is illustrated in Figure 5.1 (b). In Figure 5.1 (c), process p_0 sends a message and crashes soon afterwards. In this case, the message is not delivered to any surviving process and a new view, which excludes p_0, is delivered. Figure 5.1 (d) also demonstrates correct behaviour for this scenario, with the message delivered to processes $p_1 \ldots p_3$ before the view excluding p_0 is delivered. Figure 5.1 (e) demonstrates incorrect behaviour, with the message delivered to surviving processes in two views. Figure 5.1 (f) is also incorrect, with processes $p_1 \ldots p_3$ receiving a message from a process not in the current view.

View-synchronous group communication provides application programmers with a convenient model for maintaining replicated data. Conceptually, for processes to maintain consistent replicated data, they must behave like state machines, applying the same set of operations in the same total order to the same initial state, with a deterministic outcome. Using a view-synchronous group communication toolkit, when

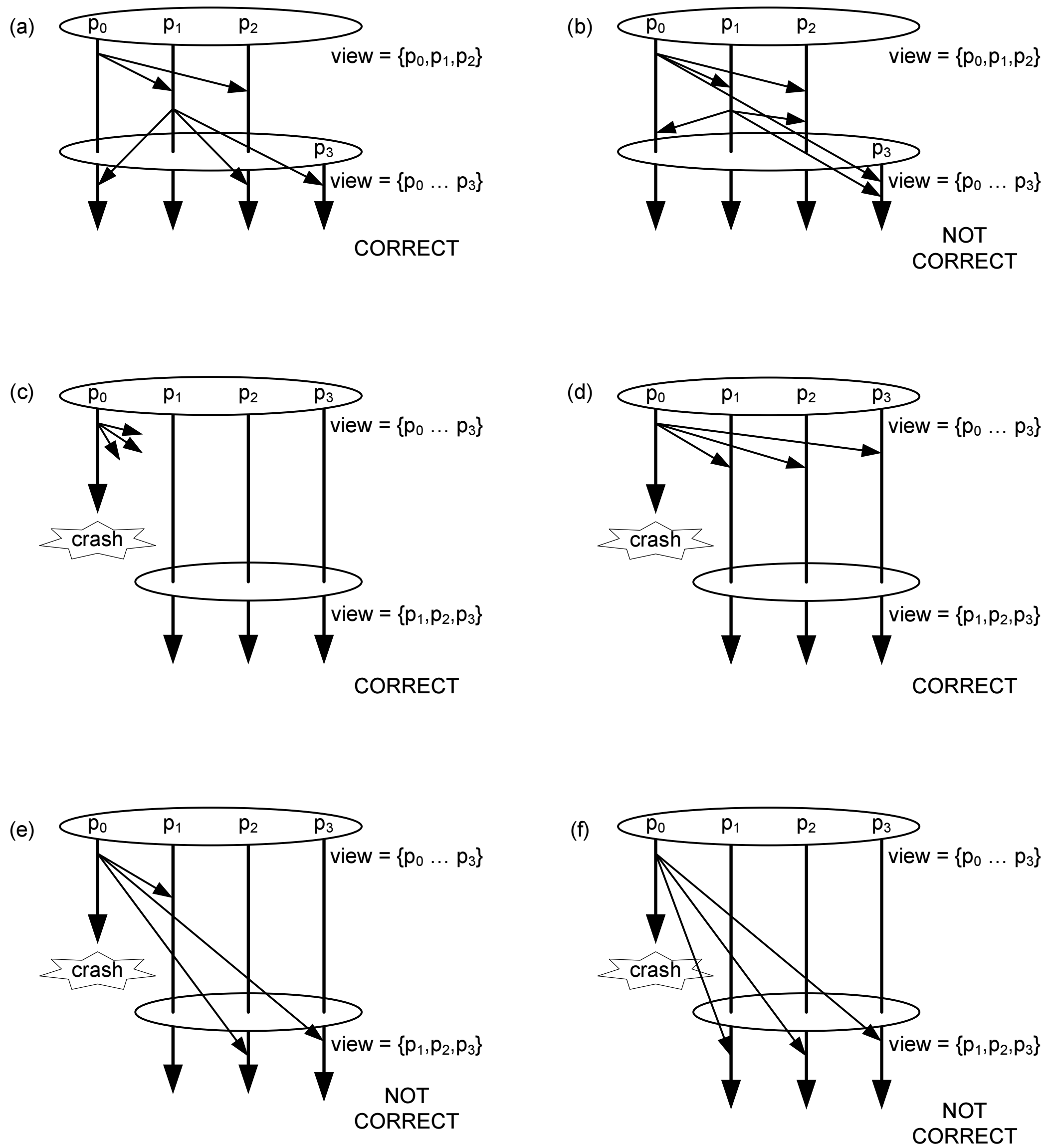


Figure 5.1: View-synchronous group communication [CDK01, Bir96]

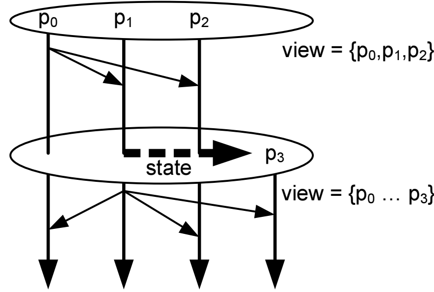


Figure 5.2: Conceptual illustration of state transfer

a new process joins the group, the current state can be copied from an existing process to the new group member during the installation of the new view (Figure 5.2). Since view-synchronous group communication guarantees that each surviving process delivers the same set of messages between any two consecutive view messages, the transferred state will reflect those operations that were applied by preexisting processes before the new view was introduced and the state was captured for transfer. After the new view has been installed, all processes, including the joining process, will resume applying the same operations to the same state.

It was stated previously that processes must apply the same operations in the same total order to the same state, to maintain consistency. Birman [Bir96] refers to this as a "closely synchronous execution". Depending on the implementation of the application, a total ordering of the operations applied by processes may not be necessary. For example, one might envisage a scenario in which the updates made by a single process can only effect a disjoint subset of the replicated state and, in a scenario such as this, FIFO ordering of messages describing the operations may be sufficient. Relaxing the ordering of multicast messages in this way allows updates issued by different processes to be interleaved, increasing concurrency. Although the processes may apply updates to the replicas in a different order, the ultimate effect of the updates will be the same as the closely synchronous execution. Birman refers to such an execution as "*virtually synchronous*".

5.1.2 The Ensemble and Maestro Toolkits

Ensemble [Hay97] is a group communication toolkit written in Objective Caml, a language which implements a dialect of the ML functional language. An important feature of the Ensemble system is its use of layered *micro-protocols* [HvR97], which when formed into a stack, provide application programmers with a mechanism to implement a group communication protocol with specific properties. For example, a protocol stack that provides virtually synchronous group communication might contain, among others, micro-protocols that provide view-synchronous communication and FIFO message ordering implemented using the UDP transport protocol.

Rather than interfacing with the Ensemble system directly, the current HammerHead prototype uses the Maestro toolkit [Vay98]. Maestro provides application programmers with an object-oriented C++ interface to Ensemble and an environment in which objects rather than processes form the members of a group. Although the object-oriented nature of Maestro is unimportant in the context of HammerHead, Maestro provides HammerHead with a client/server group communication abstraction and an implementation of a pull-style state transfer protocol for joining group members.

The basic abstraction provided by Maestro is the `Maestro_GroupMember` class, which on instantiation, is provided with a specification of the application's group communication requirements. Public methods provide the ability to join or leave the group and send point-to-point or multicast messages to the group. Protected methods, when overridden by derived classes, provide application developers with a mechanism for responding to group events – such as the receipt of a message or the delivery of a new view – in an application-specific manner.

The `Maestro_ClSv` class, derived from `Maestro_GroupMember`, provides a group communication abstraction that differentiates between *clients* and *servers* within the group. In the context of Maestro, the only difference between a server group member and a client group member is that servers maintain the application's replicated group state. `Maestro_ClSv` also provides group members with the ability to send messages to a subset of the group membership, as well as to individual members and to the whole

group. A special case of this facility is the multicast of messages to the server members of a group. The distinction between servers and clients within a group is of particular relevance to the HammerHead architecture and the use of the client/server abstraction is described later in this chapter.

While the `Maestro_ClSv` class provides application developers with the ability to implement state transfer protocols, the `Maestro_CSX` class, derived from `Maestro_ClSv`, provides developers with an implementation of a pull-style state-transfer protocol. The operation of this protocol is illustrated in Figure 5.3. Initially, the joining process p_j has server status as the sole member of its own group. When this group merges with the existing group, p_j is demoted to client status, signifying that it does not have an up-to-date copy of the existing group's state. The new member p_j requests promotion to server status in the new merged group by sending a "become server" request to the group coordinator, p_0. The coordinator installs a new state transfer view, in which p_j is designated as a joining server. During a state transfer view, only those messages that have no effect on the state and are not required by the state transfer protocol itself are sent. All other messages are queued and will eventually be sent in the next normal view[1]. In the state transfer view, p_j is prompted by the Maestro system to request the current state from an up-to-date group member – the identity of which is decided by Maestro – and p_1 is chosen in this example. Member p_1 is prompted to send its current state back to p_j, which replaces its current state with the received up-to-date state. Member p_j then informs the coordinator of the completion of the state transfer and the coordinator installs a new view in which p_j has full server status. The abstraction provided by `Maestro_CSX` handles state transfers that are interrupted and restarted, alleviating the application programmer of this responsibility.

[1]Maestro provides the application programmer with the ability to specify the messages that are allowed to be sent during a state transfer view. The behaviour described here is that used by HammerHead.

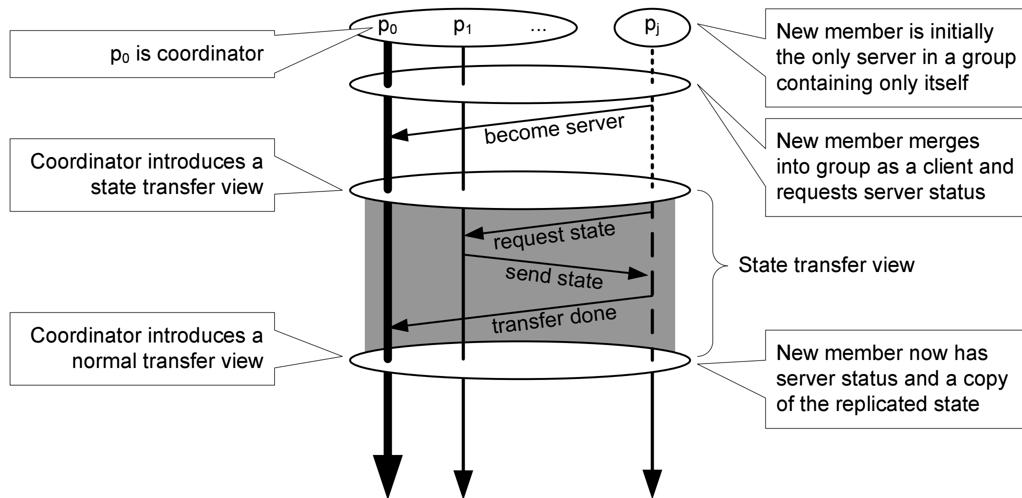


Figure 5.3: Maestro pull-style state transfer [Vay98]

5.2 Windows Media Services

Windows Media Services [GB03] is Microsoft Corporation's multimedia content distribution server. This discussion relates only to Windows Media Services 9 – the version used by the HammerHead prototype server. Windows Media Services can be used to deliver both archived on-demand content or live streams to clients. Streams can be either unicast, with an individual stream for each player, or multicast, with individual players receiving a common stream. Again, in the context of HammerHead, only the delivery of unicast on-demand streams of archived content to clients is of relevance. Unlike earlier versions, both Windows Media Services 9 and Windows Media Player 9 support the RTSP application layer protocol[2], described in Chapter 2, which is used by players to establish and control multimedia streams.

Windows Media Services has a configurable upper limit on, among other quantities, the maximum aggregate bandwidth for client streams. In addition, Windows Media Services supports multiple content mount points, called *publishing points*, each of which has its own configurable upper limits that apply only to the publishing point.

Windows Media Services provides a plug-in interface, allowing the addition of

[2]The HTTP and proprietary MMS protocols are also supported by Windows Media Services, but HammerHead relies on the use of RTSP to implement client redirection.

Event	Description
WMS_EVENT_PLAY	Notifies the plug-in when the server begins sending a stream to a client.
WMS_EVENT_STOP	Notifies the plug-in when the server stops sending a stream to a client, which may be as a result of the player stopping or pausing the stream or because the end of the stream has been reached.
WMS_EVENT_LOG	Notifies the plug-in when the server logs client activity – used by HammerHead to detect the end of a stream in circumstances not covered by the WMS_EVENT_STOP event.
WMS_EVENT_SERVER	Notifies the plug-in of the occurrence of server events, including the start and finish of content download streams.
WMS_EVENT_PUBLISHING_POINT	Notifies the plug-in of changes in the state of a publishing point, including the addition or removal of publishing points.
WMS_EVENT_PLAYLIST	Notifies the plug-in of changes related to the playing of server-side play lists, including when playback switches from one play list item to the next.

Table 5.1: Windows Media Services events captured by the HammerHead event notification plug-in

application specific functionality to the basic server. The plug-in types supported are *event notification* plug-ins, for capturing and responding to server events in an application specific manner; *authentication* and *authorization* plug-ins, allowing custom security policies to be implemented; *cache proxy* plug-ins, to allow the implementation of custom content caching solutions; *data source* plug-ins, which provide the means to retrieve multimedia content from custom network or storage sources; and *playlist parser* plug-ins, to parse custom playlist formats.

The HammerHead prototype uses a custom event notification plug-in to extract state information from a host Windows Media Services instance, capture server events and control the server instance. An event notification plug-in is a COM[3] object that implements a number of interfaces, the methods of which are invoked by Windows Media Services to notify the plug-in of the occurrence of server events. The plug-in can be enabled or disabled through a server's administration interfaces. When an event notification plug-in is started, it registers itself with the server, specifying the events that the plug-in wishes to be notified about. For example, Table 5.1 lists those events captured by the HammerHead Windows Media Services plug-in.

[3]Component Object Model

5.3 Network Load-Balancing Cluster Service

The prototype HammerHead cluster uses the Microsoft Network Load Balancing (NLB) cluster service [Mic00] to distribute client requests among cluster nodes in a statistically even manner. The NLB service operates at the device driver level and, in the configuration used by the HammerHead prototype, assigns an ISO layer-2 multicast MAC address to the network adapter that is to receive incoming client requests and assigns the same cluster IP address to each NLB cluster node. Thus, network packets sent to the cluster IP address are received by all NLB cluster nodes. The NLB driver on each node independently applies a filtering algorithm on receipt of each request and evaluates whether the packet should be passed up the local network stack for eventual delivery to an application, or discarded. The filtering algorithm will ensure that each packet is handled by exactly one node.

5.4 HammerHead Architecture

HammerHead implements a cluster-aware layer on top of a commodity multimedia streaming server, an instance of which runs on each cluster node. HammerHead consists of two main components, as shown in Figure 5.4. The first `HammerSource` component provides an abstract view of the local commodity multimedia server on each cluster node and is responsible for extracting state information from the server instance, capturing events of interest, communicating with the HammerHead cluster-aware layer and initiating the replication of existing multimedia content or the download of new content when requested by the cluster-aware layer. The `HammerServer` component is a stand-alone service that maintains the state information published by the `HammerSource` on each cluster node and updates the state in response to events, also issued by `HammerSource` instances. An aggregated cluster state is formed from the state information published by the `HammerSource` instance on each cluster node and this aggregated state is used to implement the redirection of client requests and the dynamic content replication policy. The aggregated cluster state is replicated at each

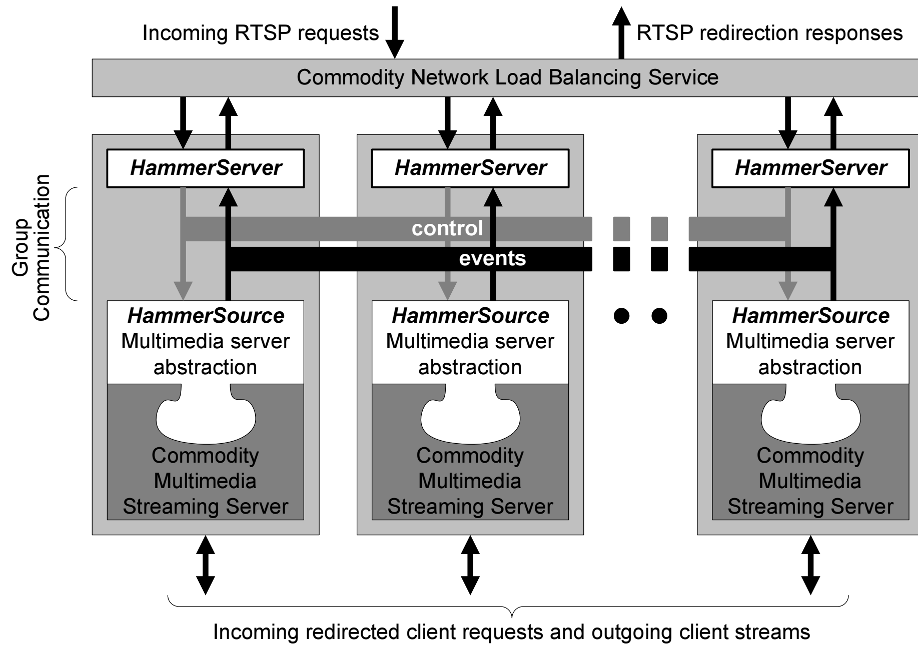


Figure 5.4: HammerHead architecture

`HammerServer` instance, allowing the workload associated with client redirection to be shared among the cluster nodes. Although Figure 5.4 shows a `HammerServer` instance on each cluster node, only one instance is required, with additional instances increasing the availability of the client redirection service and allowing the implementation of the dynamic replication policy to continue when nodes fail.

5.4.1 `HammerSource` Component

The `HammerSource` component of the HammerHead architecture provides an abstract view of the local commodity multimedia server on each cluster node. In the prototype HammerHead implementation, the multimedia streaming service is provided by Windows Media Services and the `HammerSource` component takes the form of a Windows Media Services event notification plug-in.

Node State

Before the plug-in begins processing events issued by Windows Media Services, it must capture the current state of the local cluster node. This local state information is both published for inclusion in the aggregated cluster state and maintained locally during the lifetime of the plug-in instance. The state of each cluster node is described by a hierarchy of objects that describe the local Windows Media Services instance, each of its mount points (or publishing points in the context of Windows Media Services), the replicas (files) stored under each mount point and the streams currently being provided to clients. In the HammerHead object model, `Node` objects describe those components of a HammerHead cluster that provide bandwidth and storage resources. In the simplest case, a multimedia server with a single mount point would be represented by a `SourceNode` describing the Windows Media Services instance and a `StorageNode` describing the mount point[4], as illustrated in Figure 5.5. Both `SourceNode` and `StorageNode` are subclasses of `Node`. The presentations stored under each mount point are represented by `Replica` objects, which are associated with the corresponding `StorageNode` in a hierarchical fashion. The resulting hierarchy of nodes and replicas is serialized, together with a set of `Stream` objects describing the streams that are currently active, and this serialized state is multicast to each of the `HammerServer` instances.

Streams

The `SourceNode` and `StorageNode` objects in a HammerHead cluster represent the units of bandwidth and storage provision and `Replica` objects represent the units of storage utilization. When a `Replica` is associated with a `StorageNode`, the `UsedStorage` attribute of the `StorageNode` is reduced by the file size of the presentation. Similarly, `Stream` objects represent the units of bandwidth utilization in a HammerHead cluster.

[4] `SourceNode` and `StorageNode` objects in the HammerHead object model are not equivalent to physical cluster nodes. A single cluster node will typically be represented by a `SourceNode`, corresponding to the commodity multimedia server instance, and one or more `StorageNode`s, corresponding to the server's content mount points.

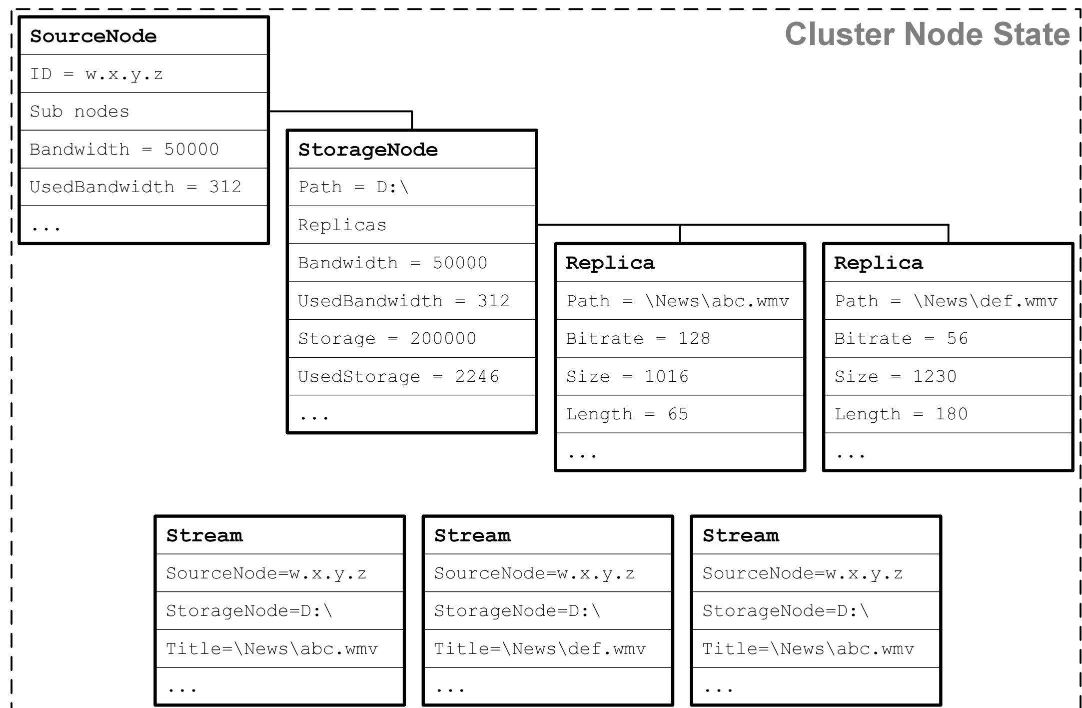


Figure 5.5: Illustration of the HammerHead state information for the Windows Media Services instance on a single cluster node

When, for example, a new `Stream` object is created to describe a client stream, the `Stream` object will identify the `SourceNode` and `StorageNode` objects that correspond to the Windows Media Services instance and mount point used to supply the stream. This set of `Node` identifiers may be thought of as a *node path*, identifying the components of the HammerHead cluster node used to supply the stream. When a stream of a presentation is started, the `UsedBandwidth` attribute of each node in the `Stream` object's *node path* is increased by the bit-rate of the presentation. Depending on the implementation of the underlying commodity multimedia server, the bit-rate of the presentation may need to be scaled by the playback rate of the stream. For example, a stream of a presentation played back by a user at twice its normal bit-rate may require twice as much bandwidth.

HammerHead assigns each `Stream` object to a stream category, based on the purpose of the stream. The stream categories supported by the prototype implementation are summarized in Table 5.2. To implement any changes required by the dynamic

Category	Description
CLIENT	A client stream served by the underlying commodity multimedia server
REPLICATION	A replication stream served by the underlying commodity multimedia server
INCOMING	A content upload stream
RESERVATION	A "dummy" stream used to reserve bandwidth for a future REPLICATION or INCOMING stream

Table 5.2: HammerHead `Stream` categories

replication policy, HammerHead requires a mechanism for both receiving and serving inter-node replication streams. In the prototype implementation, this functionality is provided by Windows Media Services, which provides a mechanism for downloading new content identified by a URI. The REPLICATION and INCOMING stream categories are used to identify the streams at the source and destination ends of an inter-node replication stream respectively. It may be desirable to reserve the bandwidth required for an inter-node replication stream, temporarily reducing the bandwidth available to clients and ensuring that there will eventually be sufficient bandwidth to begin replication. HammerHead provides the RESERVATION "dummy" stream category for this purpose. Streams in the RESERVATION category compete with CLIENT streams for bandwidth, but INCOMING and REPLICATION streams can use the bandwidth that has been reserved. Although the HammerHead prototype was configured to give replication streams priority over client streams, the use of RESERVATION streams may vary between implementations.

`HammerSource` Events

Once a `HammerSource` plug-in has started and the local node state has been multicast to the cluster's `HammerServer` instances, events describing any subsequent changes to the node's state are multicast to the `HammerServer` instances, allowing the aggregated state to be updated over time. For example, when a Windows Media Services instance begins sending a stream to a client, a STREAM_START event, containing a `Stream` object describing the stream, is multicast by the local `HammerSource` plug-in and the `HammerServer` layer will update the aggregated state accordingly, increasing

Event	Description
NODE_ADD	A node object (e.g. a new content mount point) has been added to an existing SourceNode.
NODE_REMOVE	A node object (e.g. an existing content mount point) has been removed from a SourceNode.
REPLICA_ADD	A Replica has been added to a StorageNode.
REPLICA_REMOVE	An existing Replica has been removed from a StorageNode.
STREAM_START	A stream has started.
STREAM_STOP	A stream has stopped.
REPLICA_STATE_CHANGE	The state of a Replica has changed.
REPLICA_UNDELETE	A Replica that was scheduled for deletion has been undeleted.

Table 5.3: HammerHead events issued by HammerSource plug-ins to the HammerServer cluster-aware layer

the UsedBandwidth attribute of the associated nodes and adding the Stream object to the set of streams currently being served. The events issued by HammerSource plug-ins in the prototype cluster are summarized in Table 5.3.

Rather than issuing events as they occur, HammerSource plug-ins collect events into batches and multicast these batches of events to the HammerServer layer periodically (e.g. every two seconds in the prototype cluster). Issuing batches of events in this manner reduces the overhead incurred by the group communication service, particularly when the actions taken by a HammerSource plug-in result in a sequence of events over a short period of time, as will occur during the creation of new replicas, for example, which is discussed later.

Replica States

The lifetime of a HammerHead replica is described by a sequence of state transitions, illustrated in Figure 5.6. When a change of Replica state occurs, the HammerSource hosting the Replica multicasts a REPLICA_STATE_CHANGE event to the HammerServer instances.

Consider, for example, the case where a HammerSource is requested (by a HammerServer instance) to create a new replica of a presentation by downloading it, either from another cluster node or from a remote location. The HammerSource initially creates a new Replica object in the *inactive* state and sends a request, to a remote

multimedia server instance, to begin the download. When the download stream for the new replica begins, the `Replica` object enters the *active* state. Once the replica has been successfully downloaded, the `Replica` object enters the *ready* state, indicating that the replica can now be used to service client requests. The replica will remain in the *ready* state until the `HammerSource` is requested (by a `HammerServer` instance) to delete it. If the download stream for a new replica is stopped or fails, the `Replica` object enters the *stalled* state[5].

The `HammerSource` component provides a locking mechanism for `Replica` objects, to support the creation of new replicas. Consider, for example, the case where the dynamic replication policy dictates that a replica should be moved from cluster node A to cluster node B. If concurrent requests were issued by a `HammerServer` instance to the `HammerSource` plug-ins on both nodes, there would be no guarantee that the replica would be copied successfully to B before it is removed from A. In circumstances such as these, the corresponding `Replica` object on A would be locked until the successful download of the replica on B is confirmed (by its transition to the *ready* state). When a request is received to delete a `Replica` object with one or more locks, the `Replica` enters the *delete scheduled* state, indicating that the replica will eventually be deleted but can still be used to service client streams. When the number of locks on the `Replica` object reaches zero, the replica still cannot be deleted until any streams currently being served by the replica have terminated. In this case, the `Replica` object remains in the *delete pending* state until all existing streams have terminated. New client requests will not be redirected to a replica in the *delete pending* state. Finally, after the last stream ends, the `Replica` object enters the *can delete* state and can either be removed immediately by the `HammerSource` or, depending on the policy implemented by cluster, left on the file system until the `HammerSource` is instructed by a `HammerServer` instance to delete it[6] or until additional storage capacity is required.

[5]`Replicas` in this state can only be removed, but the state serves to provide `HammerServer` instances with information about the failure of the download, allowing a replacement replica to be created, as described later in this chapter.

[6]Requiring a `HammerServer` instance to confirm the removal of a replica in this manner would give the cluster-aware layer the opportunity to confirm that removing the replica will not result in the loss of the presentation, which may occur in exceptional circumstances (e.g. when nodes fail) due to the

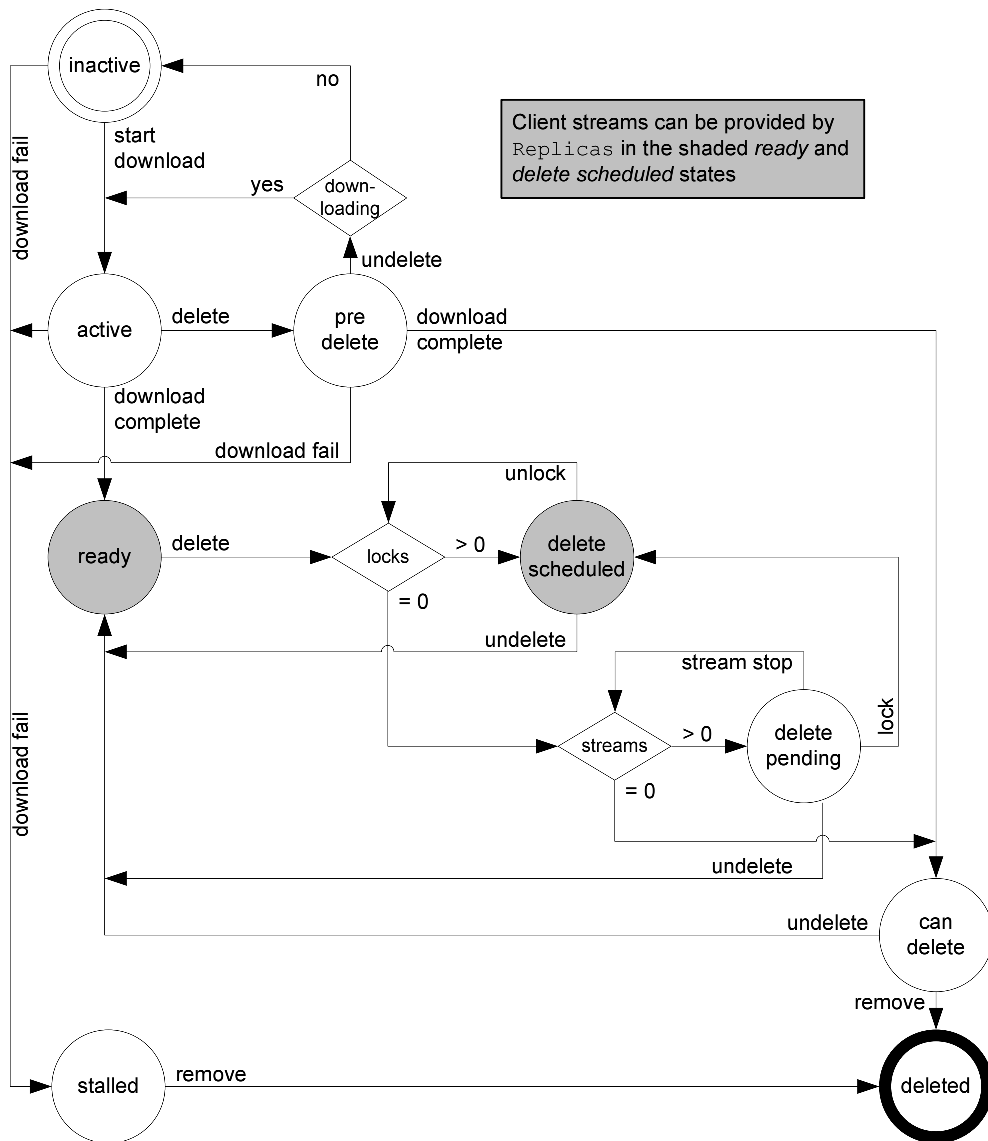


Figure 5.6: HammerHead `Replica` state transitions

A `HammerSource` may be requested to "undelete" a `Replica` object that was previously marked for deletion. If the `Replica` object was previously in the *ready* state, it immediately reenters that state. Otherwise, it reenters either the *active* or *inactive* state depending on the `Replica` object's original state when deleted.

The addition of new replicas under a mount point of an active Windows Media Services instance can result either from the download of a new replica or from the placement of the replica under the mount point by some external agent (e.g. a user moving a file with a command shell). In both cases, the `HammerSource` must be able to detect the arrival of the file under the mount point and decide whether the file represents a replica that can be used to supply on-demand multimedia streams. The prototype HammerHead implementation uses the `ReadDirectoryChanges(...)` Win32 API function to receive notifications of changes to the directory structure of each of the mount points. When the addition of a new file is detected, the `HammerSource` searches for a corresponding `Replica` object in the *inactive* state. If a matching `Replica` is found, the file is assumed to have resulted from the start of a download stream of the file and the `Replica` enters the *active* state. Otherwise, if no `Replica` is found, it is assumed that the file has been added by an external agent, a new `Replica` is created and the `Replica` immediately enters the *ready* state. Similar actions are taken when files are removed from a mount point. The renaming of a file is treated as the removal of one file and the addition of another.

5.4.2 `HammerServer` Component

The `HammerServer` component of the HammerHead architecture implements the cluster-aware layer at the core of the clustered multimedia server, redirecting client requests to appropriate commodity multimedia server instances, implementing the dynamic content replication policy and managing the creation, movement and removal of replicas. Each `HammerServer` instance is a stand-alone service that maintains a replica of the

asynchronous nature of communication in HammerHead.

aggregated state of each Windows Media Services instance in the cluster. The aggregated state is updated in response to the receipt of HammerHead events, issued by HammerSource plug-ins, as described earlier.

Presentation Objects

When a HammerSource joins a HammerHead cluster and its serialized state is received by a HammerServer, the state is incorporated into the aggregated cluster state. Replica objects that correspond to replicas of the same presentation are associated with a common Presentation object. Presentation objects describe presentations that can be streamed to clients and each Presentation must have one or more associated Replicas. Thus, while each Replica corresponds to a physical file, Presentation objects are meta-data objects used to associate replicas of the same presentation with each other.

HammerHead requires a convention to associate replicas with Presentation objects. The prototype implementation makes the assumption that replicas accessed by clients using the same path and file name, on different Windows Media Services instances, are replicas of the same presentation, regardless of replicas' absolute filesystem paths and filenames. For example, consider two replicas stored on different cluster nodes, both of which are accessed by clients using the following URI:

```
RTSP://<node IP address>/News/2004/abc.wmv
```

Both of the corresponding Replica objects would be associated with a common Presentation object, which would be given the identifier /News/2004/abc.wmv. Clients requesting streams of the presentation from a HammerHead cluster, identified by the cluster DNS name hmrhead.cs.tcd.ie, would supply the URI:

```
RTSP://hmrhead.cs.tcd.ie/News/2004/abc.wmv
```

in an RTSP request. When adding a Replica to the aggregated cluster state, a HammerServer will first search for an existing Presentation whose identifier matches the Replica. If one is found, the Replica is associated with the existing Presentation, otherwise a new Presentation is created.

Handling Client Requests

All HammerServer instances share the workload associated with the redirection of client requests to appropriate cluster nodes. This is achieved by using a commodity hardware or software network load balancing solution to distribute incoming client RTSP requests among HammerServer instances, with each instance handling a disjoint subset of the requests. The prototype HammerHead cluster uses the Microsoft Network Load-Balancing cluster service, which was described earlier, in section 5.3.

On receipt of an RTSP request, a HammerServer instance will parse the request and extract the path identifying the requested presentation. If a HammerServer cannot find a Presentation object with an identifier matching the request URI, an RTSP 404 (File Not Found) response is returned to the client. Otherwise, if a matching Presentation is found, the list of Replicas associated with the Presentation is examined and the most suitable cluster node with a replica of requested presentation is selected to handle the request. The most suitable cluster node must satisfy the following conditions:

- The Replica hosted by the cluster node must be in either the *ready* or *delete scheduled* states.
- The unused bandwidth of each Node object required to supply a stream of the replica (e.g. the SourceNode and StorageNode objects in the cluster configuration illustrated in Figure 5.5) must be greater than or equal to the bandwidth required by the stream.
- The cluster node must be the most suitable candidate node (i.e. Windows Media Services instance), according to the load-balancing policy implemented by the cluster. In practice, the load-balancing policy will be dependant on the node and storage topology of the cluster. For example, in the basic cluster configuration described later in Chapter 6, the most suitable candidate node is the one associated with the StorageNode with the lowest proportional bandwidth utilization.

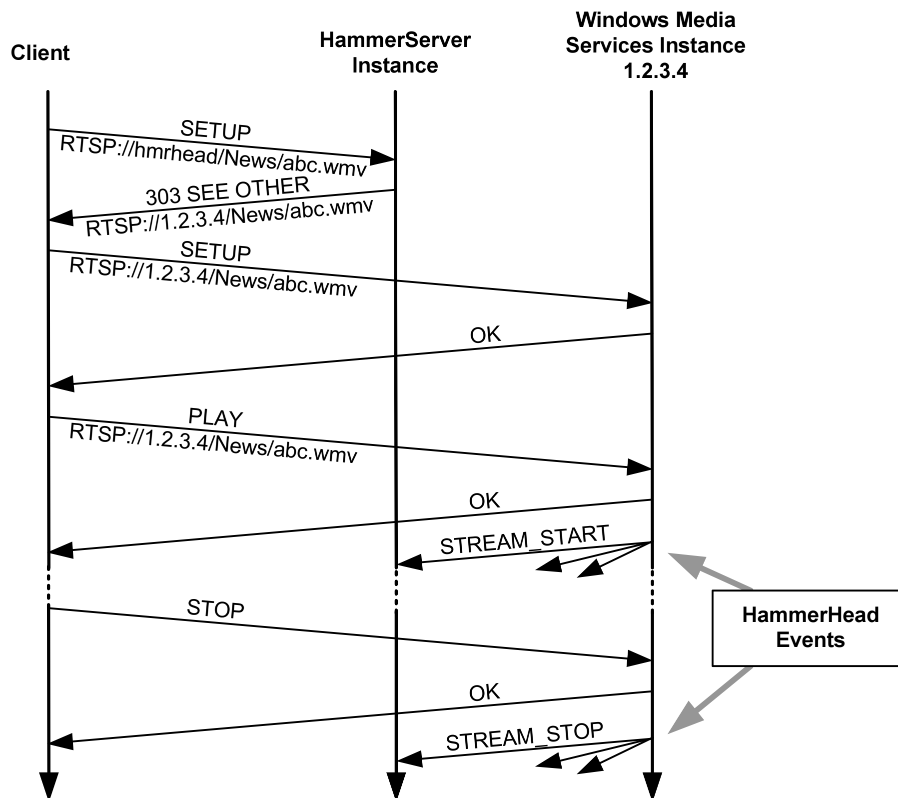


Figure 5.7: HammerHead client redirection

If there is no `Replica` that satisfies these conditions, an RTSP 453 (Not Enough Bandwidth) response is returned to the client. If a suitable `Replica` is found, the `HammerServer` instance generates an RTSP 303 (See Other) response with a target URI that identifies the selected cluster node and the path to the replica on that node. The `HammerServer` layer plays no further active role in the client's interaction with the multimedia server (although it will learn about the client's streaming activity through events issued by `HammerSource` plug-ins). The exchange of RTSP requests and responses and the HammerHead events resulting from the start and end of the associated stream are illustrated in Figure 5.7. This redirection scheme allows unmodified commodity multimedia players to access the service provided by the cluster and only requires a client to be redirected once during a streaming session.

Estimating Demand

The scheme described in Chapter 4 for estimating the relative demand for each multimedia presentation requires the total duration of all streams of each presentation to be measured over a period of time. HammerHead records the total duration of streams of the same presentation as attributes of the `Presentation` objects that represent the presentations available to clients. When a client stream is started on a Windows Media Services instance, the local `HammerSource` plug-in multicasts a STREAM_START event, which contains a `Stream` object describing the stream, to every `HammerServer` instance. On receipt of a STREAM_START event, each `HammerServer` instance records the current time in the `Stream` object. Similarly, when a stream ends and a STREAM_STOP event is issued by a `HammerSource`, each `HammerServer` subtracts the start time recorded in the `Stream` object from the current time, yielding the duration of the stream. The stream duration is added to the cumulative stream duration for the presentation, which is recorded as an attribute of the corresponding `Presentation` object. Before evaluating the relative demand for each presentation at the end of an evaluation period, HammerHead adds the current play duration of all active streams – up to the end of the evaluation period – to the total duration of their corresponding presentations and resets the start time of each stream to the current time. In other words, the total playback duration for a presentation in a single evaluation period will include the duration of partial streams, as well as streams that both started and finished in the same period, as illustrated in Figure 5.8.

Implementing Dynamic Replication

After evaluating the relative demand for each presentation, as described above, a coordinating `HammerServer` instance will use the Dynamic RePacking algorithm, described in Chapter 4, to determine a suitable assignment of replicas to nodes or, more precisely, `Replica` objects to `StorageNode` objects. (The choice of the coordinating `HammerServer` is determined by the Ensemble group communication toolkit and is described later in section 5.4.3.) Although only one `HammerServer` instance actively

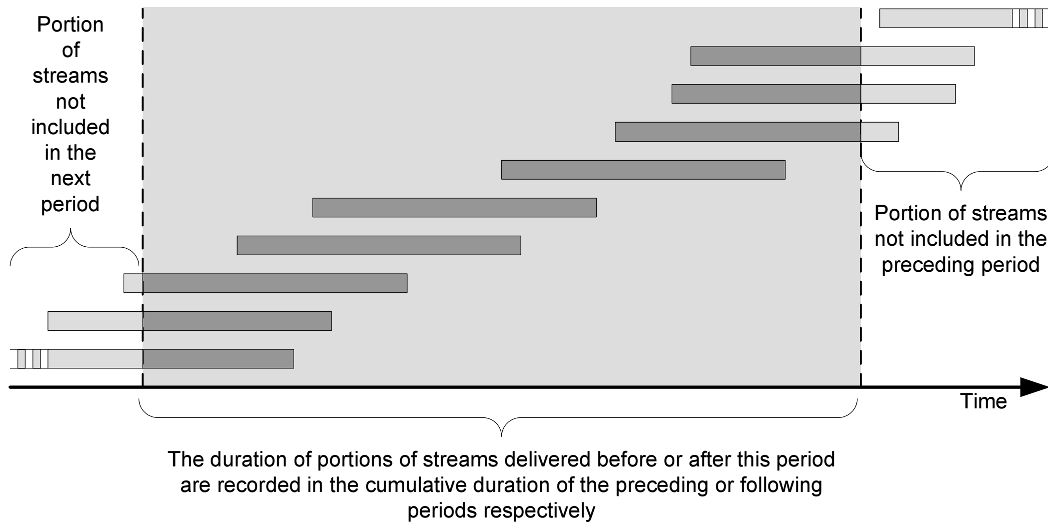


Figure 5.8: Measurement of stream duration

implements the Dynamic RePacking policy, every other instance passively evaluates the relative demand of each presentation, allowing the coordinating instance to be replaced if it fails or is removed from the cluster.

After updating the estimated relative demand for each presentation, the coordinating `HammerServer` takes a snapshot of the cluster state, which is used as the input to the Dynamic RePacking algorithm. The result of the algorithm is a set of changes to the current assignment of replicas to cluster nodes, corresponding to the difference between the two assignment matrices described in Chapter 3. Each of the changes will require either the creation of a new replica or the removal of an existing one. The changes are implemented by sending CREATE_REPLICA and REMOVE_REPLICA control messages to the `HammerSource` plug-ins on the cluster nodes that are to add and remove replicas respectively.

Figure 5.9 shows the sequence of control messages issued by a `HammerServer` to create a single replica and the resulting events issued by the `HammerSource` plug-ins hosting the source and destination replicas for the replication stream. Each CREATE_REPLICA control message is accompanied by a template `Replica` object, describing the new replica to be created. As well as describing the new replica, the template

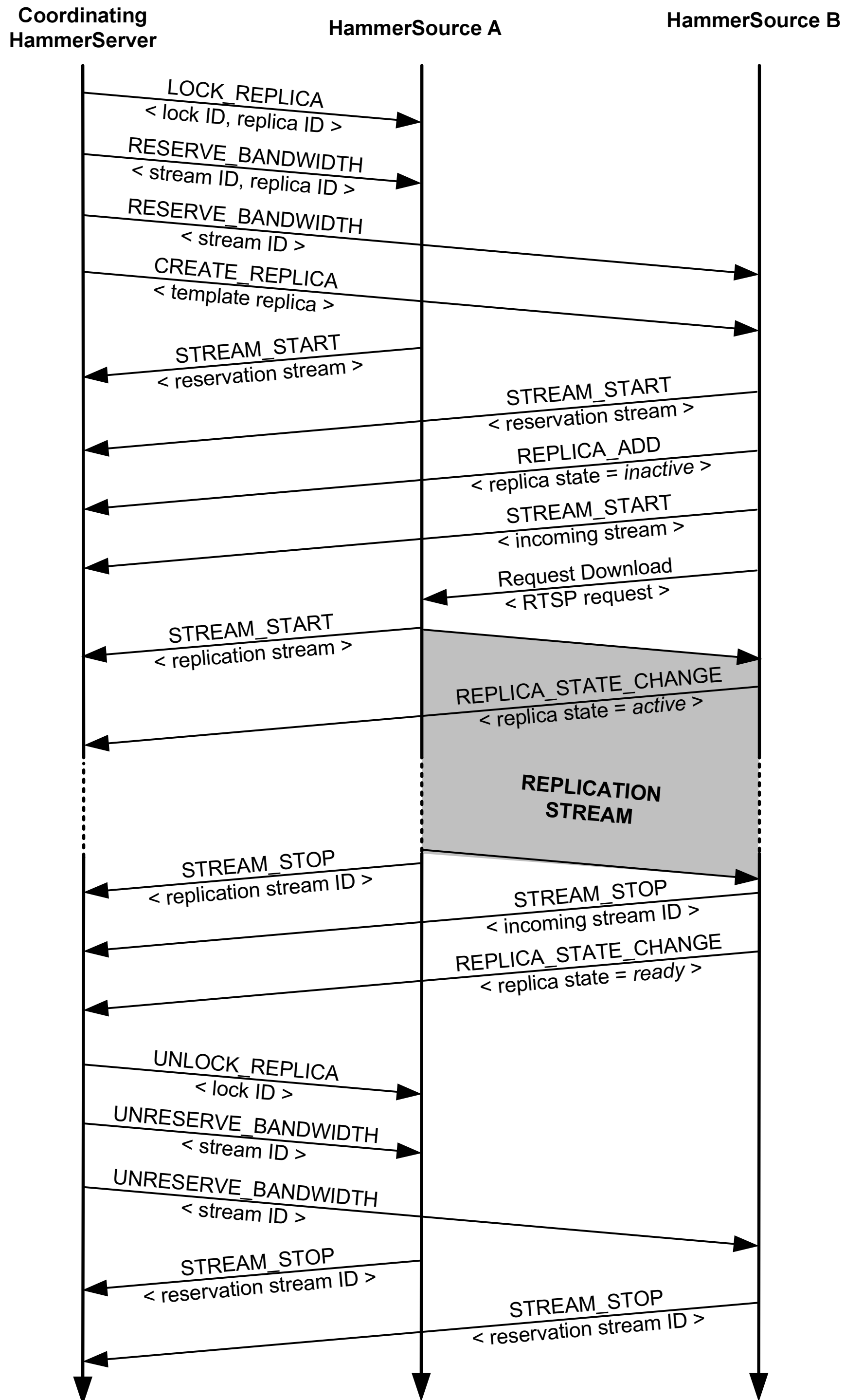


Figure 5.9: HammerHead replica creation

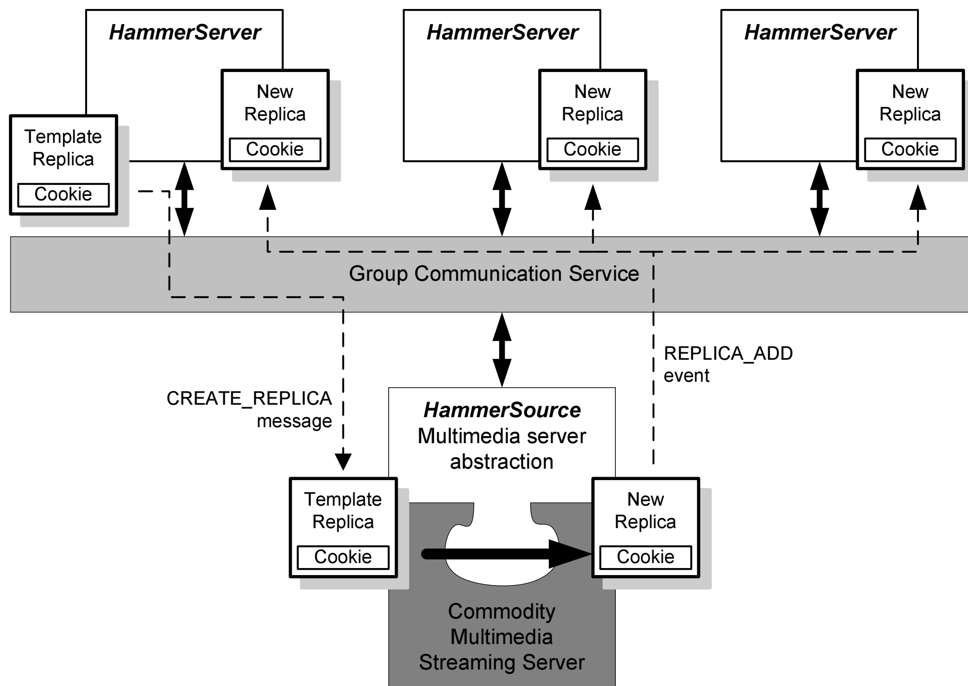


Figure 5.10: Template `Replica` and new `Replica` objects containing "cookie" information

`Replica` serves as a "cookie", in which the coordinating `HammerServer` can store information related to the creation of the replica. When the `HammerSource` plug-in creates the new `Replica` object (initially in the *inactive* state), the resulting REPLICA_ADD event will pass the "cookie" information to each `HammerServer` instance, as shown in Figure 5.10, allowing one of the other instances to take over the role of coordinator, should the original coordinator fail.

The information contained in the template `Replica` includes a URI, identifying the location from which the new replica should be downloaded. For inter-node replication streams, the URI will identify a replica of the presentation on another cluster node, although replicas may also be downloaded from a location outside the HammerHead cluster. By requiring the coordinating `HammerServer` to specify the source for the replication stream, the selection of the source is subject to the same load-balancing constraints as other client streams.

Before sending a CREATE_REPLICA control message, the coordinating HammerServer must lock the source Replica that will supply the replication stream and reserve resources for both the REPLICATION and INCOMING streams, and this is done by sending LOCK_REPLICA and RESERVE_BANDWIDTH control messages to the source and destination HammerSource plug-ins. The LOCK_REPLICA message increments the lock count on a Replica object. RESERVE_BANDWIDTH messages instruct HammerSource plug-ins to create new Stream objects with the RESERVATION stream type. Both locks and RESERVATION streams have a limited life span and must be periodically refreshed by the coordinating HammerServer, until the new replica has been downloaded. RESERVE_BANDWIDTH control messages contain the identifier for the Stream object that will eventually be issued with the STREAM_START event by a HammerSource plug-in, allowing the coordinating HammerServer to create a unique identifier, which is stored in the cookie contained in the template Replica. Similarly, the coordinating HammerServer creates a lock identifier, which is also stored in the cookie. Thus, both RESERVATION streams and locks are associated with the creation of a single, specific replica. Should the coordinator fail during the creation of the new replica, the new coordinator can use the information in the cookie to continue refreshing the locks and RESERVATION streams associated with each *inactive* and *active* Replica, preventing them from timing out. In the exceptional case where a coordinating HammerServer requests a lock and reserves bandwidth on a source Replica object, but fails before sending a CREATE_REPLICA message to the destination HammerSource, the lock and reservation stream will time-out, releasing the lock and releasing the reserved bandwidth resources.

Once the replication stream has ended and the new replica has been successfully created, the corresponding Replica object will enter the *ready* state. On detecting the transition to the *ready* state, the coordinating HammerServer (which need not be the coordinator that originally initiated the replication) can issue an UNLOCK_REPLICA message to unlock the source replica. Similarly, UNRESERVE_BANDWIDTH messages sent to both the source and destination HammerSource plug-ins will remove the

RESERVATION streams, releasing the reserved bandwidth to service subsequent client requests.

If a presentation, of which there is only a single replica stored in the cluster, is to be moved from one location to another, it is important that the `HammerSource` plug-in hosting the corresponding `Replica` object receives the LOCK_REPLICA message before the REMOVE_REPLICA message, to prevent the removal of the replica before the inter-node replication stream has started. The FIFO message ordering enforced by the Ensemble group communication toolkit will ensure that, if the REMOVE_REPLICA message is sent after the LOCK_REPLICA message, the messages will be delivered in that order, causing the `Replica` object to be locked first. When the REMOVE_REPLICA control message arrives, the `Replica` will enter the *delete scheduled* state, and will eventually proceed to the *delete pending* and *can delete* states, once the replication stream has ended, the `Replica` has been unlocked and any active client streams have ended.

Although a replica whose corresponding `Replica` object is in the *delete pending*, *delete scheduled*, *can delete* or *delete after create* state may be undeleted, a coordinating `HammerServer` makes no assumption about the ability of the `HammerSource` to undelete the replica. It instead assumes that the replica will need to be created by a replication stream and sends a CREATE_REPLICA control message as before. On receiving the CREATE_REPLICA message, if the `HammerSource` is able to undelete the replica, it will do so, returning the `Replica` to the state it was in before being deleted. A REPLICA_UNDELETE event containing the template replica, which was issued by the `HammerServer` in the CREATE_REPLICA message, is multicast by the `HammerSource`, allowing the coordinating `HammerServer` to release any locks and stop any RESERVATION streams that were created to facilitate the replication stream.

Inter-node replication streams may fail for a number of reasons:

- There are insufficient resources available on the cluster node that is to store the new replica.

- There are insufficient resources available on the cluster node that is to supply the replication stream.
- Another error causes the stream to fail.

In each of these cases, the `Replica` object associated with the replica being created enters the *stalled* state, from which it can only be deleted. Entering the stalled state, however, allows the coordinating `HammerServer` to take remedial action. When a `HammerServer` instance receives a REPLICA_STATE_CHANGE event signalling that a `Replica` has entered the *stalled* state, the `Replica` is placed on a list of stalled `Replica` objects. Periodically, the coordinating `HammerServer` processes this list, taking remedial action for each stalled replica. In the current prototype, the only action taken is to send a REMOVE_REPLICA message to the `HammerSource` hosting the stalled `Replica` object and to attempt to create a new replica of the same presentation on the same node. Creation of the new replica may be impossible if, for example, the only cluster node with a replica of the presentation has failed, in which case the `HammerServer` aborts its attempts and the presentation will no longer be available to clients.

5.4.3 Group Communication in HammerHead

HammerHead uses the Ensemble group communication toolkit [Hay97] to implement the communication channel between `HammerServer` and `HammerSource` instances. The Maestro interface to Ensemble is used to provide HammerHead with the `Maestro_CSX` abstraction of client and server group members and with a pull-style state transfer protocol. A single HammerHead group is created, of which all `HammerServer` and `HammerSource` instances are members. Within this group, `HammerSource` instances are Maestro clients and can communicate with all members in the group but do not participate in the Maestro state transfer protocol. `HammerServer` instances are Maestro servers and use the Maestro state transfer facility to provide new instances with an up-to-date copy of the aggregated cluster state.

Although Maestro and Ensemble allow any group member to communicate with

any other group member – either with point-to-point messages or with multicast messages to some or all group members – HammerHead restricts the communication channels available to `HammerServer` and `HammerSource` instances. Specifically, `HammerSource` instances may only send multicast messages to the entire set of server group members (the `HammerServer` instances). In contrast, `HammerServer` instances may only send point-to-point messages to individual client group members (`HammerSource` instances). Accordingly, all `HammerServers` will receive a common stream of messages from the `HammerSources` in the group, while each `HammerSource` will receive its own set of messages from the `HammerServer` layer.

FIFO multicast message ordering is sufficient since there is no communication between individual `HammerServer` instances or individual `HammerSource` instances, and since the events issued by each `HammerSource` only apply to the portion of the aggregated state published by that instance. Thus, although different `HammerServer` instances may see different cluster states at different times, and the events from different `HammerSource` plug-ins may be processed in a different order at each instance, the eventual effect of the events received by each instance will be the same.

The coordinating HammerServer discussed in section 5.4.2 is always the first Maestro server in the group view delivered by the Maestro toolkit. Since only the coordinating `HammerServer` plays an active role in implementing the Dynamic RePacking algorithm, the `HammerSource` plug-ins in a group will only receive point-to-point messages from a single `HammerServer` in any given view.

When a new group view is introduced by Maestro at a `HammerSource` plug-in, if the union of the sets of Maestro servers in the old and new views is equal to the null set (i.e. there are no servers carried over from the previous view), the `HammerSource` plug-in captures its current state – as described in section 5.4.1 – and multicasts the serialized state to the `HammerServer` instances. Each `HammerServer` will independently merge the serialized state into the aggregated cluster state. It should be noted that this same action is performed regardless of whether a `HammerServer` instance is started and joins a group containing only existing `HammerSource` plug-ins, or a `HammerSource`

plug-in is started and joins a group containing existing `HammerServer` instances. In both cases, a `HammerSource` plug-in will see `HammerServer` instances joining a group that did not previously contain any server members.

When a `HammerSource` leaves a HammerHead group and a new view that excludes the `HammerSource` is introduced at every group member, the `HammerServer` instances will remove the part of the aggregated state associated with the `HammerSource`. This is done in a structured manner, by emulating STREAM_STOP, REPLICA_REMOVE and NODE_REMOVE events, thus stopping any active streams served by the departing cluster node, and removing any `Replica` and `Node` objects hosted by the plug-in.

`HammerServer` State Transfer

HammerHead uses the pull-style state transfer facility provided by the `Maestro_CSX` class, which was described in section 5.1.2. When a `HammerServer` merges into a group containing one or more `HammerServer` instances, Maestro will determine the direction of the state transfer based on the approximate age of each member – with newer members receiving the cluster state from older ones – and install a state transfer view as shown previously in Figure 5.3. A `HammerServer` selected to receive the cluster state from another member will be requested by Maestro to ask an existing member to transfer the state. The `HammerServer` formulates a state request message, which it submits to Maestro. Maestro selects an existing member to supply the state and forwards the state request message to it.

On receiving a state request message, a `HammerServer` takes a snapshot of the current HammerHead state. The state could be returned directly to the joining member as a Maestro message. To avoid the overhead incurred by sending large messages in Maestro, however, the prototype implementation transfers the state "out-of-band" with respect to the group communication service, as recommended in [Bir96].

The joining member, on receiving the state snapshot, deserializes and installs the cluster state. Once the state has been installed, the joining member informs Maestro and a new normal view is installed by the group's coordinator. Once the new normal

view has been installed, any messages stalled during the state transfer view can be sent and will be delivered to both old and new `HammerServer` instances.

If the HammerHead cluster state is large, it may be undesirable to stall the sending of messages for the time taken for the joining member to obtain and install the state information. A modified state transfer protocol might block messages only until the current state has been captured, at which point any blocked messages could be sent. A protocol such as this, however, would require the joining member to queue any messages delivered to the group until the captured state has been transferred and installed. Issues relating to the transfer of large quantities of state information are discussed in [Bir96].

5.5 Summary

This chapter has presented the HammerHead multimedia server cluster architecture and has described the key technologies on which the prototype implementation depends. HammerHead implements a cluster-aware layer on top of a commodity multimedia server, allowing the service provided by the commodity server to scale beyond the capacity of a single node. A `HammerSource` plug-in captures the state of the commodity server on each cluster node and publishes that state in the aggregated cluster state maintained by each `HammerServer` instance. `HammerSource` instances keep the published state up-to-date by issuing event messages. The aggregated cluster state is used by `HammerServer` instances to redirect initial client requests and to implement the Dynamic RePacking policy described in Chapter 4.

The use of a group communication system provides HammerHead with a means of managing the membership of a cluster and reliably maintaining the replicated cluster state. The communication model used by HammerHead, which is based on asynchronous message passing rather than a request-response model, and in which `HammerServer` instances communicate though `HammerSource`s, rather than directly with each other, increases concurrency and simplifies the fail-over of the coordinator role when

`HammerServer` instances fail or are removed from the cluster.

Finally, by using resource reservation, locking and replica state transitions, the creation and removal of replicas can be tightly coordinated with the provision of client streams and any incomplete changes in the distribution of replicas can be taken into account by future invocations of the Dynamic RePacking algorithm.

Chapter 6

HammerHead Performance

Evaluating the performance of any multimedia server can be expensive, both in terms of time and material resources. The high network, processor and memory resource requirements for a single multimedia stream means that any performance evaluation will be coarse-grained in comparison with other applications evaluated over the same time and with the same resources.

For example, consider the evaluation of the performance of a server with a bandwidth capacity of 200Mbps supplying multimedia streams each with a data rate of 400Kbps. The server can supply an average of five hundred such streams concurrently. If the average duration of each stream is thirty minutes, the server can accept just over sixteen new requests each minute, whereas a World-Wide Web server with the same specification could service many times that number of requests in the same period. In addition, there will be a delay of thirty minutes – the average duration of the streams – before the multimedia server is fully utilized and the performance can be measured, disregarding any time required by the server to adapt to the characteristics of the supplied workload. In contrast, the short request service time of a World-Wide Web server would allow it to reach full utilization almost immediately.

Obtaining results for a broad range of multimedia server configurations and request patterns is a lengthy process, particularly when measuring the effects of heterogeneous presentation characteristics and changing popularity distributions. Obtaining

real-time results from a multimedia server for a single data point can take many hours. The collection of the performance data presented in this chapter alone takes twelve days if the experiments are run consecutively, without interruption. Although all existing dynamic replication policies in the literature appear to have been evaluated through simulation studies, it was felt that evaluating the real-time performance of a live system was an important part of the verification of both the Dynamic RePacking algorithm and the HammerHead architecture. A combination of both real-time analysis of a prototype server cluster and compressed-time event-driven simulation has, therefore, been used to evaluate the performance of the HammerHead architecture and the Dynamic RePacking content distribution policy. In this chapter, the methodology and results for both the prototype and simulation experiments are presented.

6.1 Performance Metrics

The following metrics are used to evaluate the performance of the HammerHead architecture:

Achieved load Most of the experiments described below generate a client workload with a mean request arrival rate chosen to achieve a desired target load. The achieved load is the proportion of the maximum cluster service capacity used during the test. Thus, if the target load is 0.5, the achieved load should also be 0.5, under ideal circumstances. The upper limit on the achieved load for a cluster is 1.0. Achieved load reflects only the load on the cluster resulting from streams supplied to clients, and excludes the load resulting from the overhead of creating and moving replicas within the cluster. As a result, the cost of performing dynamic replication is reflected in the achieved service load.

Yield As described in Chapter 2, yield expresses the number of accepted requests as a proportion of the number of requests submitted to the service. When the target service load is 1.0, yield will be approximately equal to achieved load.

Load imbalance To maximize performance under high load conditions, the cluster workload should be distributed evenly across all nodes. Under lower load conditions, when the target load is easily achieved and performance is less dependent on the distribution of replicas, load imbalance is a more suitable measure of server performance and gives a useful indication of how the cluster will cope with increasing load. The degree of load imbalance in the HammerHead prototype has been determined by measuring the standard deviation between the load on each node. If the load on each node is identical and the cluster is perfectly load-balanced, the standard deviation will be zero.

Storage utilization The aggregate storage utilization of the cluster is the sum of the storage required by the presentations stored on each of the cluster's nodes. The lower limit on the required storage capacity is the sum of the data size of each presentation in an archive, without any replication (i.e. there is a single replica of each presentation):

$$\text{minimum storage requirement} = \sum_{i=1}^{K} V_i \tag{6.1}$$

where V_i is the on-disk data size of each presentation. A cloned server cluster with N nodes requires the highest storage capacity since every node must store a replica of every presentation:

$$\text{cloned storage requirement} = N \times (\text{minimum storage requirement}) \tag{6.2}$$

Unless otherwise stated, storage utilization is expressed as a proportion of the minimum storage requirement in the experiments described below.

Replication bandwidth The relative popularity of presentations changes over time and a HammerHead cluster must react to these changes by creating additional replicas of some presentations, removing existing replicas or moving replicas from one node to another. The creation and movement of replicas requires the use of

server bandwidth that could otherwise be used to supply client streams and a single inter-node replication stream uses resources on both the source and destination nodes of the stream. A HammerHead cluster should perform enough redistribution of replicas to achieve load-balancing, without excessively reducing the bandwidth available to supply client streams. In particular, the bandwidth used by replication should be low during periods when the relative demand for each presentation is static. Measuring the bandwidth used for the creation and movement or replicas allows the cost of a particular replication policy to be compared with its benefits, in terms of achieved service load. Replication bandwidth can be presented as an absolute value or as a proportion of the used bandwidth of the cluster, and both measures have been used.

Replication rate When combined with the measured replication bandwidth, the number of replication streams started per hour allows further conclusions to be drawn about the benefit that results from the use of valuable server resources for the creation and movement of replicas. Replication rate is independent of the bit-rate of each replication stream.

6.2 HammerHead Prototype Performance Evaluation

The performance of the HammerHead architecture using the Dynamic RePacking replication policy has been evaluated by constructing a prototype four-node HammerHead server cluster and generating a simulated client workload. Although the pattern of client requests is synthetic, the requests and resulting streams are real and, from the perspective of the HammerHead server cluster, are indistinguishable from the requests received from real users. The experiments conducted were designed to evaluate:

1. The performance of the prototype HammerHead cluster and the Dynamic RePacking policy, compared with that of a cloned server cluster using complete replication.

2. The performance of the prototype for varying target service loads.

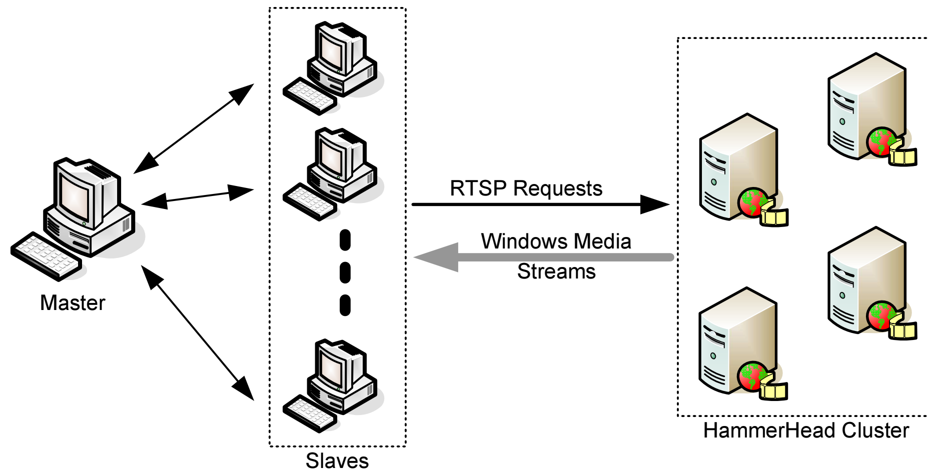


Figure 6.1: Master-slave architecture of workload generator

3. The effect of periodic changes in the relative popularity of presentations.

4. The effect of reducing the available storage capacity.

5. The performance of the prototype when supplying streams of presentations with varying bit-rates and durations.

6. The performance of the prototype when node failures occur.

Multimedia Workload Generation

To test the prototype HammerHead multimedia server cluster in a controlled manner, a distributed client workload generator was developed. The workload generator is based on Microsoft Windows Media Player technology and can submit requests for multimedia streams to any service based on Windows Media Services technology, such as the HammerHead server cluster.

A single host was incapable of consuming a sufficient number of concurrent multimedia streams to fully utilize the prototype HammerHead server cluster in the test environment described here. Instead, multiple client hosts were used to generate the workload and a master-slave model was adopted to coordinate the efforts of the hosts. Figure 6.1 illustrates the architecture of the HammerHead workload generator.

The master workload generator generates requests by randomly selecting a URI from a configurable list. The interarrival times for requests are generated as random variates from a exponential distribution, with a configurable mean arrival rate. Each URI is selected with a probability determined by the Zipf distribution with $\theta = 0$. Thus, the probability of requesting URI i is given by:

$$p_i = \frac{1}{i\left(\sum_{j=1}^{K} \frac{1}{j}\right)} \tag{6.3}$$

where K is the number of URIs that can be selected, as described in Chapter 2. When generating a request, the master selects a URI at random as described, selects the least loaded slave workload generator and sends the URI to the slave.

On receipt of a URI from the master, a slave creates a new Windows Media stream reader object using the Microsoft Windows Media Format SDK [Mic]. From the perspective of the server cluster, the reader object is no different from a Windows Media Player instance. To reduce the overhead of consuming the stream on the workload generator slaves, however, the reader object has no user interface and does not decompress or render the streamed packets – the packets are simply discarded when they arrive. When a slave starts a stream or a stream ends, the slave informs the master generator of the event and also of the slave's new current load, allowing the master to approximately balance the workload across the set of slaves.

The master generator can be configured to periodically "shuffle" the URI request probabilities, as described in Chapter 2. The URIs are shuffled in the manner used by Chou et al. [CGL00]. To simulate sudden increases in the popularity of presentations, the least popular presentation becomes the most popular and the popularity of the remaining presentations is gradually reduced. More formally, after every shuffle period has expired, the probability of requesting each presentation i is reassigned as follows:

$$p_i = \begin{cases} p_0 & \text{if } i = K \\ p_{i+1} & \text{otherwise} \end{cases} \tag{6.4}$$

Each of the experiments described below required configuring the workload generator to achieve a desired target service load, expressed as a proportion of the maximum cluster bandwidth. The mean request arrival rate, λ, required to achieve a target service load is determined as follows:

$$\lambda = \frac{T}{\sum_{i=1}^{K} p_i W_i B_i} \tag{6.5}$$

where λ is the required mean request arrival rate, T is the target cluster bandwidth in megabits per second, K is the number of presentations that can be requested and p_i, W_i and B_i are the request probability, playback duration and bit-rate of presentation i respectively.

Prototype HammerHead Configuration

A four-node HammerHead server cluster was configured as illustrated in Figure 6.2. Each node contained a single AMD Athlon MP 1900+ processor and 1GB[1] of RAM, and ran the Microsoft Windows Server 2003 operating system.

Each node had two 100Mbit network adapters, the first of which was used for the sole purpose of receiving initial RTSP requests and redirecting the clients to suitable Windows Media Services instances. These initial client requests were distributed among server nodes using the Microsoft Network Load Balancing (NLB) cluster technology [Mic00], which was briefly described in Chapter 5. Clients were then redirected (by `HammerServer` instances) to the Windows Media Services instance on the second network adapter of the selected node. Redirected clients contacted the Windows Media Services instance directly and began receiving the requested stream, as described in Chapter 5.

Since Microsoft recommends configuring Windows Media Services with an upper

[1]Initially, the experiments were conducted with 512MB of RAM installed in each node. It was observed, however, that when a node approached the limit of its network bandwidth capacity (50Mbps) excessive "thrashing" was taking place. This "thrashing" resulted from the high memory utilization of the Windows Media Services process and resulted in requests for streams occasionally timing out. Although the number of requests that timed out was small in relation to the total number of requests submitted, the rate was variable and had a small but unpredictable effect on the measured performance of the HammerHead system. It is recommended that a server running Windows Media Services have at least 50% more memory than is required by the Windows Media Services process [Fer03]. This recommendation was satisfied by increasing the RAM in each HammerHead node to 1GB.

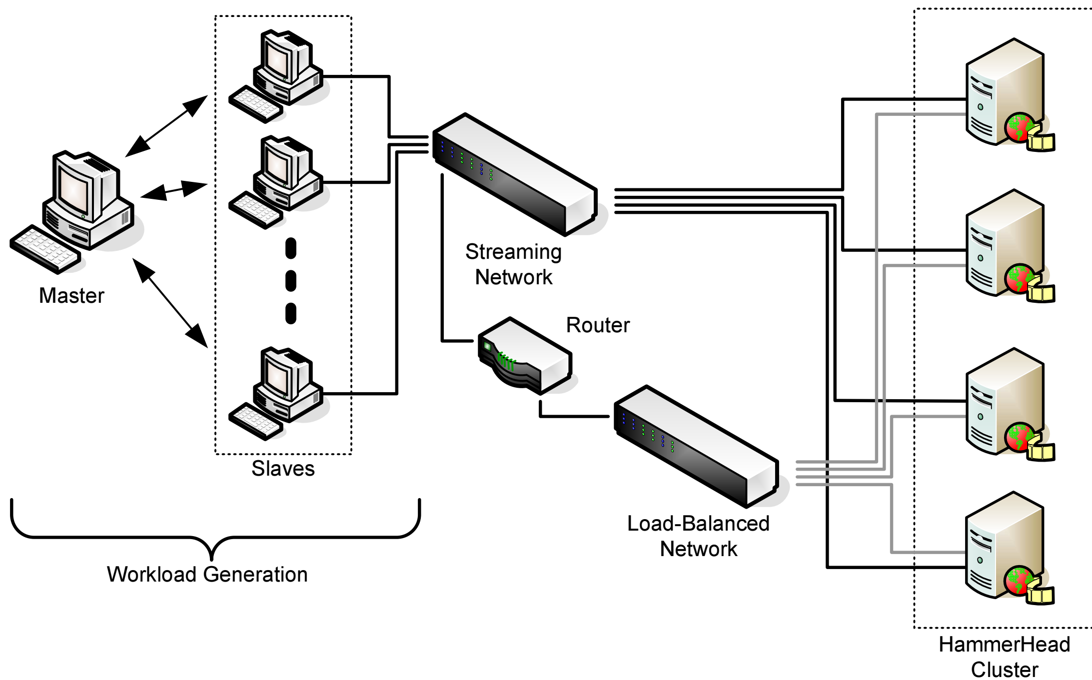


Figure 6.2: HammerHead performance evaluation environment

bandwidth limit for multimedia streaming of one half of the capacity of the network adapter [Fer03], the bandwidth limit of each cluster node was set to 50Mbps. Thus, the maximum allowable bandwidth for the four node cluster was 200Mbps. Each Windows Media Services instance had a single content mount point, also with a bandwidth limit of 50Mbps.

Ensemble was configured to communicate and perform membership detection on the second adapter, since it could not co-exist with the NLB cluster technology on the first adapter. Both client streams and inter-node replication streams were also transmitted on the second adapter. As a result, the existence of either an incoming or outgoing replication stream on a node reduced the capacity of the node to supply client streams.[2]

Certain HammerHead configuration parameters were common to each of the experiments performed. In each case, the rate of replication was 1.0, meaning that the

[2]An alternative approach could use either the adapter on the load-balanced network, if permitted by the load-balancing solution, or a separate dedicated adapter, to perform inter-node replication.

time taken to create or move a replica was the same as the playback time of the corresponding presentation. Thus, an incoming or outgoing replication stream consumed the same network bandwidth as a client stream of the same presentation. The `HammerServer` instances were configured to reevaluate the distribution of replicas every five minutes and the demand for each presentation was averaged over six evaluation periods, using the weighted average scheme described in Chapter 4. Unless otherwise stated, fault-tolerant Dynamic RePacking was used in each experiment and was configured to distribute replicas to maintain load-balancing after a single node failure. The fit threshold described in section 4.2.5 was not used in the experiments on the prototype HammerHead cluster but was evaluated in the simulation study, which is described later, in section 6.3.

The statistics used to evaluate server performance were collected on a periodic basis every thirty seconds. With the exception of Prototype Experiment 4, which evaluates the effect of constrained storage capacity on the server, the storage capacity of each node was sufficiently large to store the entire archive on every node and the results presented show what proportion of this available capacity was actually required by each configuration.

6.2.1 Prototype Experiment 1: Replication Policy

This experiment evaluates the performance of the HammerHead prototype and, in particular, compares the performance of selective replication using Dynamic RePacking with that of a cloned server cluster, in which every presentation is replicated on every node. The results demonstrate that dynamic replication approaches the performance of a cloned server cluster, while significantly reducing the required storage capacity.

A set of one hundred presentations was generated with the properties listed in Table 6.1, along with the parameters used to generate the simulated workload. With the exception of the cloned configuration (described below), a single replica of each presentation was initially assigned to a randomly selected node in the four-node cluster,

Configuration:	MinOne	Top10	Top50	MinTwo	Cloned
Presentation properties					
Duration (sec)	758				
Size (MB)	23				
Data rate (Kbps)	272				
Number of presentations	100				
Experiment parameters					
Target load	1.0 (200,000Kbps)				
Requests per hour	3,492				
Total requests	27,936 (8 hours)				
Shuffle period	no shuffle				
HammerHead Configuration					
Minimum replica count	1	2	2	2	4
Minimum count proportion	1.0	0.1	0.5	1.0	1.0
Cluster storage capacity (MB)	9,200	9,200	9,200	9,200	9,200

Table 6.1: Parameters used for the replication policy experiment

before enabling the HammerHead service (the `HammerServer` component) and Windows Media Services on each node. The same random assignment was used for this experiment and for each of the other prototype experiments. The mean request arrival rate was chosen to attempt to fully utilize the cluster, representing a target load of 1.0. The evaluation of each cluster configuration lasted eight hours and the performance during the first hour was ignored, to allow the service time to adapt to the client request pattern. The relative popularity of each presentation remained the same for the duration of the experiment.

To demonstrate the trade-off between storage capacity utilization and performance, a number of different Dynamic RePacking policy configurations were evaluated. The first, referred to as *MinOne*, created only enough additional replicas of each presentation to provide load balancing and to maintain load-balancing after a single node failure using fault-tolerant Dynamic RePacking. In other words, the minimum number nodes to which each presentation was assigned was one (corresponding to a *minimum replica count* of 1 and a *minimum count proportion* of 1.0). The second configuration, *Top10*, created at least two replicas of the most demanding 10% of the presentations (corresponding to a *minimum replica count* of 2 and a *minimum count proportion* of

0.1). Similarly, *Top50* created at least two replicas of the most demanding 50% of the presentations (corresponding to a *minimum replica count* of 2 and a *minimum count proportion* of 0.5). The final Dynamic RePacking policy configuration, *MinTwo*, assigned a replica of each presentation to at least two cluster nodes (corresponding to a *minimum replica count* of 2 and a *minimum count proportion* of 1.0). The results of the experiment are shown in Figure 6.3. The performance of a cloned server cluster, in which the multimedia archive was replicated on every node, with Dynamic RePacking disabled, but still using `HammerServer` instances to implement client redirection, was also evaluated.

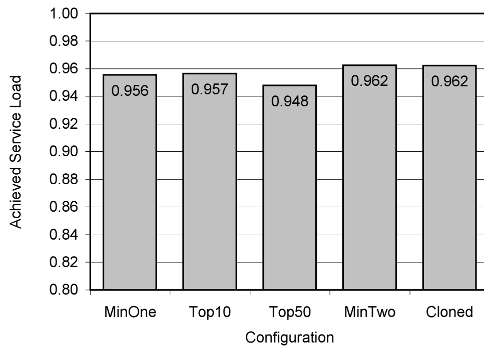
Cloned **Server Cluster**

It is clear from the results obtained that the cloned server cluster achieved the best multimedia streaming performance, with an achieved load of approximately 0.962 for a target load of 1.0. The cloned server cluster also exhibited the lowest load imbalance of all of the evaluated configurations.

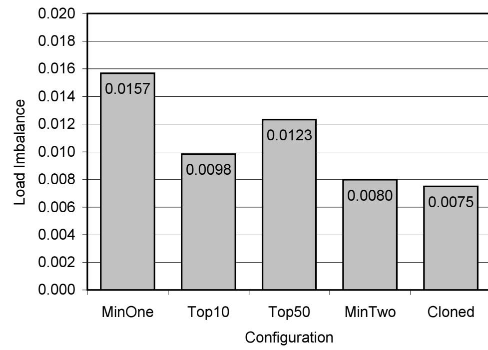
The performance achieved by this cloned cluster configuration is the optimal performance for the prototype HammerHead server cluster and provides a suitable baseline against which the performance of the remaining configurations can be measured. This configuration, however, also has the highest storage capacity requirement (400% of the minimum capacity required by the archive) since each node was required to store a replica of the entire archive of one hundred presentations.

MinOne **Configuration**

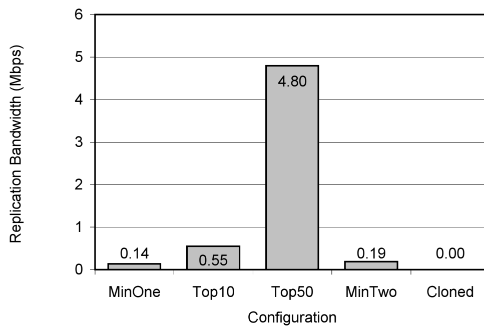
The storage capacity requirement for the first server configuration using the Dynamic RePacking policy, referred to as *MinOne*, was approximately 2406MB, or 105% of the storage capacity required to store a single replica of each presentation (2300MB) and just over 25% of the capacity required by the cloned server cluster. Despite a significant reduction in the number of replicas of each presentation, this configuration still performed well with respect to achieved load and load imbalance. The achieved



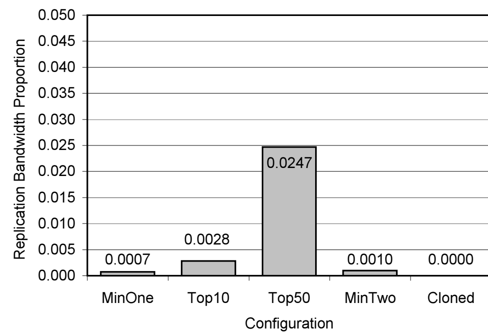
(a) Achieved Load as a Proportion of Maximum Load



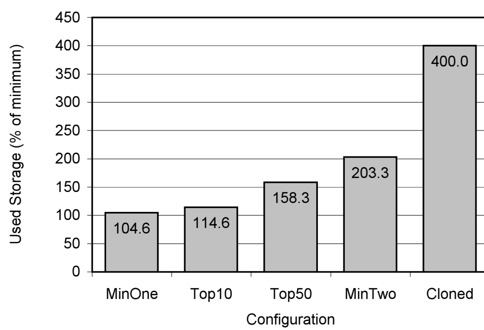
(b) Load Imbalance



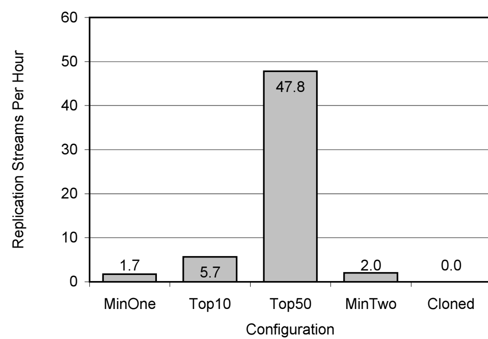
(c) Replication Bandwidth



(d) Replication Bandwidth as a Proportion of Total Cluster Bandwidth



(e) Used Storage (% of Minimum Required)



(f) Replication Streams Started per Hour

Figure 6.3: Results of the replication policy experiment

load was approximately 0.956, 0.6% less then the optimal performance achieved by the cloned server cluster. Although the average load imbalance was twice that of the cloned configuration, it represents relatively small differences between the load on each node. This can be illustrated by calculating the average difference between the number of streams on the most heavily loaded and least heavily loaded nodes over the duration of the experiment. In this case, the average difference was 7.3 streams and the average number of concurrent streams served by each cluster node was 176.

Implementing Dynamic RePacking requires the distribution of replicas to be reexamined periodically (every five minutes in this case) and, as a result of this reevaluation, some replicas may need to be created, removed or moved between nodes. The cluster resources required by the resulting inter-node replication streams reduces the resources available to service client requests, so it is important that the cost of implementing dynamic replication is low. The average server bandwidth required by replication streams for this cluster configuration, including both incoming and outgoing streams, was approximately 137Kbps. This represents approximately 0.07% of the total bandwidth used by the cluster for both client streams and inter-node replication streams. There were just under two new inter-node replication streams started every hour. The period between successive invocations of Dynamic RePacking was deliberately chosen to be small in these experiments. In practice, if the relative popularity of presentations was known to change gradually over long periods, the period between successive reevaluations of the distribution of replicas could be increased, reducing the average number of inter-node replication streams.

The network overhead required by the use of group communication, including both the traffic generated by the group communication toolkit itself and the messages transmitted by HammerHead using the group communication system, was approximately 2.7KB/sec for the *MinOne* configuration when the cluster was fully utilized. With no client workload applied to the cluster, the group communication overhead was approximately 1.4KB/sec. The size of the state information transferred to a joining `HammerServer` instance, when the cluster was fully loaded during this experiment, was

approximately 490KB. Using commodity compression tools, this could be significantly compressed – to less than 21KB in this case – reducing the time required to transfer the state to the new group member. The total execution time of the Dynamic RePacking algorithm (including each of the iterations required by fault-tolerant Dynamic RePacking) for this configuration was approximately 47ms.

Top10 Configuration

Configuring the Dynamic RePacking policy to create at least two replicas of the most demanding 10% of the presentations yielded an achieved load of 0.957, which is approximately the same as that achieved by the *MinOne* configuration, despite the creation of additional replicas. The average load imbalance, however, was significantly reduced. The storage utilization increased to 115% of the minimum, which is still significantly less than that of the cloned configuration.

A notable feature of the results for this configuration is the increase in the rate of inter-node replication, with the proportion of bandwidth required for replication increasing by a factor of four over that for the *MinOne* configuration. This is explained by the frequent changes in the minimum number of replicas required for each presentation, which results from changes in their relative ranking by demand. For example, the presentation ranked tenth requires two replicas while the presentation ranked eleventh requires only one and, since the difference between number of requests for these presentations is relatively small (they will account for 1.9% and 1.7% of the requests, according to the Zipf distribution), their relative rankings will change frequently. These results suggest that while the relative demand for presentations can be measured over the short term, the ranking of presentations should be based on data gather over a longer period.

Top50 Configuration

The *Top50* server configuration, which created at least two replicas of the most demanding 50% of the presentations, might be expected to have achieved better performance

than either the *MinOne* or *Top10* configurations. The results, however, show a decrease in achieved load to 0.948 and an increase in the rate of inter-node replication to almost 2.5% of the total bandwidth used by the cluster. Like the *Top10* configuration, this is due to frequent changes in the relative ranking of presentations and the resulting changes in the minimum number of replicas required for each presentation. This effect is more pronounced in the *Top50* configuration due to the increase in the number of replicas created and the decrease in the difference between the relative popularity of lower-ranked presentations. The storage utilization for this configuration is still significantly less than that for the cloned cluster configuration, at 158% of the minimum capacity required to store a single replica of each presentation.

Like the *Top10* configuration, these results suggest that the ranking of presentations should be based on data gathered over a longer period. An additional experiment, which ranked each presentation according to the number of streams served since the beginning of the experiment, but which still performed replication to achieve load-balancing based on the demand estimated over the shorter term (as in previous experiments), produced a significant improvement in performance. The bandwidth used to perform inter-node replication was reduced to 0.5Mbps and the achieved service load was 0.963.

***MinTwo* Configuration**

The final configuration examined in this experiment, referred to as *MinTwo*, created at least two replicas of each presentation, incurring an increase in storage utilization to 203% of the minimum required, or approximately half that of the cloned server configuration.

Unlike the *Top10* and *Top50* configurations, each presentation was always assigned to at least two cluster nodes, so changes in the relative ranking of presentations by demand have a significantly reduced impact on server performance. The achieved load was 0.962, the same as that achieved by the cloned cluster configuration. The average load imbalance was also significantly reduced and approaches that achieved by

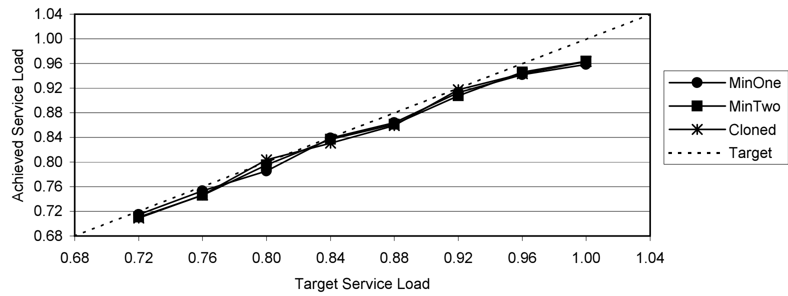
the cloned configuration. The rate of inter-node replication was similar to that of the *MinOne* configuration.

6.2.2 Prototype Experiment 2: Target Cluster Load

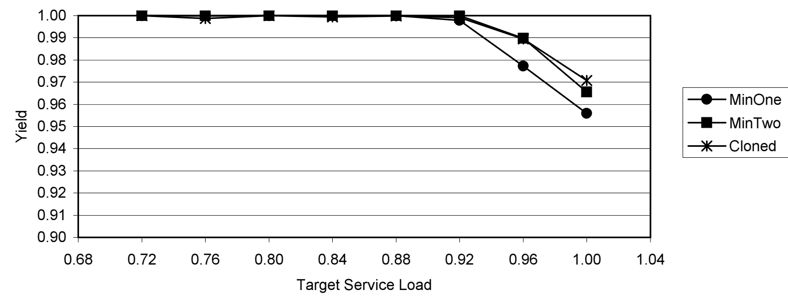
The preceding experiment compared the performance of different Dynamic RePacking policy configurations with that of a cloned server cluster for a target load equal to the maximum allowed service load. This second experiment evaluates the performance of the prototype HammerHead server cluster for lower target service loads. The results show that Dynamic RePacking can adapt to client behaviour before the cluster is fully utilized and that Dynamic RePacking yields the same performance as a cloned server cluster for all but the highest target service loads.

The performance of the server cluster, using the *MinOne*, *MinTwo* and *Cloned* cluster configurations from the previous experiment, has been evaluated and the parameters for the experiment are shown in Table 6.2. The set of presentations used was the same as that used in the preceding experiment and the relative popularity of each presentation remained the same for the duration of the experiment. Since a total of 24 different data points needed to be obtained, it was impractical to run each combination of cluster load and Dynamic RePacking policy configuration for the same eight hour period used in Experiment 1. Instead, each combination was evaluated over a three hour period, with measurements from the first half hour ignored to allow the server to stabilize. To reduce the time required for the server to adapt to the generated workload, each policy configuration was "warmed up" for a one hour period. After this one hour period, during which the cluster adapted to the pattern of requests, the placement of replicas on nodes was recorded and was restored after each change of target load. For each combination of target load and Dynamic RePacking policy configuration, the average achieved service load, yield and load-imbalance were evaluated and the results are summarized in Figure 6.4, with the broken line in Figure 6.4 (a) representing the target load.

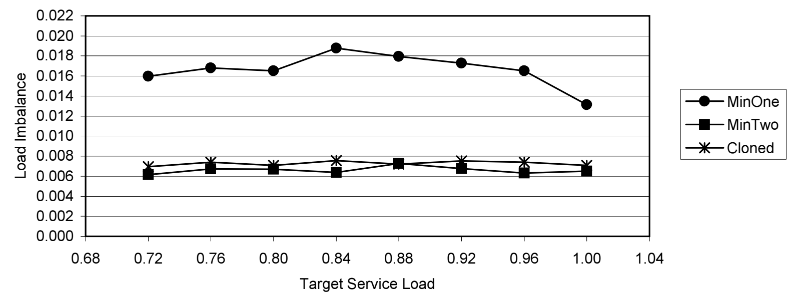
The short duration of this experiment, relative to the duration of the previous



(a) Achieved Load as a Proportion of Maximum Load



(b) Yield



(c) Load Imbalance

Figure 6.4: Results of the target cluster load experiment

Configuration:	MinOne	MinTwo	Cloned
Presentation properties			
Duration (sec)	758		
Size (MB)	23		
Data rate (Kbps)	272		
Number of presentations	100		
Experiment parameters			
Target load	0.72, 0.76, 0.80, 0.84, 0.88, 0.92, 0.96, 1.00		
Requests per hour	2,514, 2,654, 2,794, 2,933, 3,073, 3,213, 3,352, 3,492		
Total requests	7,542, 7,962, 8,382, 8,799, 9,219, 9,639, 10,056, 10,476		
Shuffle period	no shuffle		
HammerHead Configuration			
Minimum replica count	1	2	4
Minimum count proportion	1.0	1.0	1.0
Cluster storage capacity (MB)	9,200	9,200	9,200

Table 6.2: Parameters used for the target cluster load experiment

experiment, increased the experimental error. It is still clear from the results, however, that each configuration approximately achieved the target load, up to target loads of 0.96, when the achieved load began to deviate from the target load as more requests were rejected. In fact, by analyzing the statistics gathered by the workload generator, rather than by the HammerHead `HammerServer` component, the achieved yield (requests accepted as a proportion of requests submitted) for each configuration can be determined and these results are shown in Figure 6.4 (b). Almost 100% yield was maintained for target loads of up to 0.92, with a small decrease in yield as the target load increased further. The increase in performance of the *MinTwo* configuration over *MinOne* is clear, although *MinOne* still achieved over 95% yield for a target load of 1.0. The results show that load imbalance remains approximately static for each configuration, regardless of target load, demonstrating that Dynamic RePacking can adapt to client behaviour under lower load conditions.

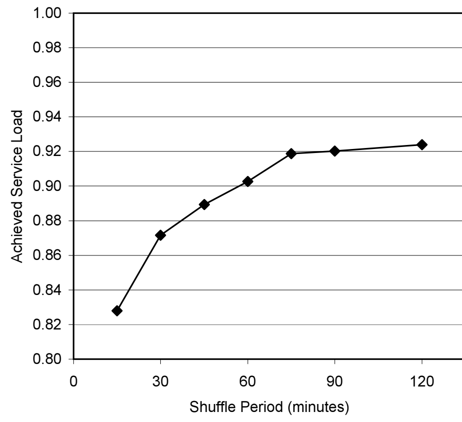
6.2.3 Prototype Experiment 3: Changing Presentation Popularity

The relative popularity of the presentations made available to clients is likely to change over time. A multimedia server cluster based on selective replication, such as HammerHead, must therefore continuously or periodically reevaluate the relative demand for each presentation and adapt to any changes in client behaviour. To determine the effect of such changes in popularity on the performance of the prototype HammerHead cluster, the performance of the *MinOne* policy configuration used in previous experiments has been evaluated for varying popularity "shuffle" frequencies. The results demonstrate that, although sudden changes in client behaviour temporarily reduce the performance achieved by the prototype cluster, the Dynamic RePacking policy quickly adapts to the new behaviour.

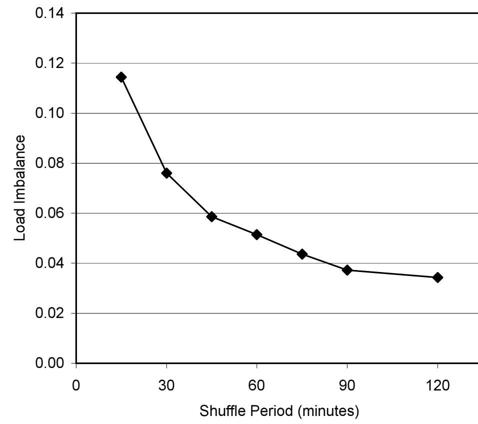
The same set of one hundred homogeneous presentations used in the two preceding experiments was also used for this experiment and the target cluster load was again 1.0. The relative popularity of the presentations requested by the workload generator were periodically shuffled to simulate sudden changes in the demand for individual presentations[3], as described in section 6.2. The period between successive popularity shuffles ranged from fifteen minutes to 120 minutes. For each shuffle frequency, server performance was measured over an eight hour period and the data from the first shuffle period was excluded from the results to allow the cluster time to adapt to the initial request pattern. The parameters for this experiment are shown in Table 6.3 and the results are summarized in Figure 6.5.

The results show that performance, with respect to achieved load, load imbalance and replication bandwidth, improves as the period between successive popularity shuffles increases. The decrease in storage capacity utilization is explained by the decreased inter-node replication rate: moving a replica from one node to another temporarily doubles the storage required for that replica, since the old replica cannot be removed until the new replica has been created. The difference between the highest

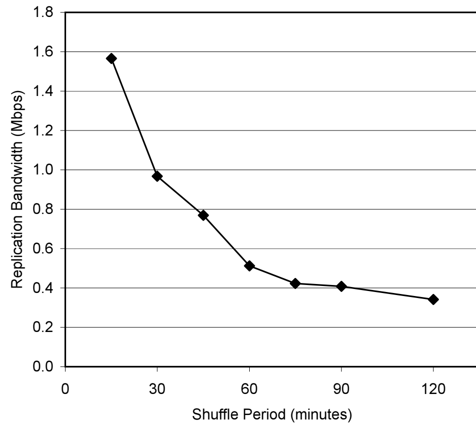
[3] Such changes may be due to a sudden increase in the demand for an existing presentation, or the addition of a new presentation with high demand.



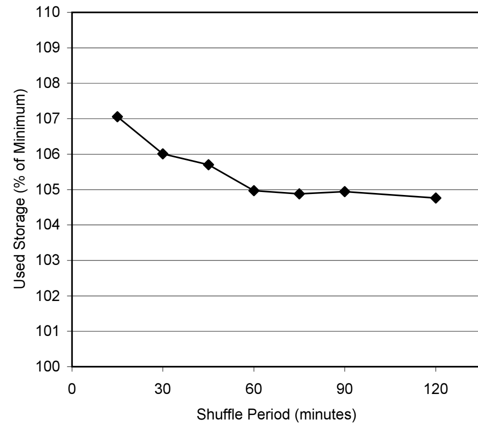
(a) Achieved Load as a Proportion of Maximum Load



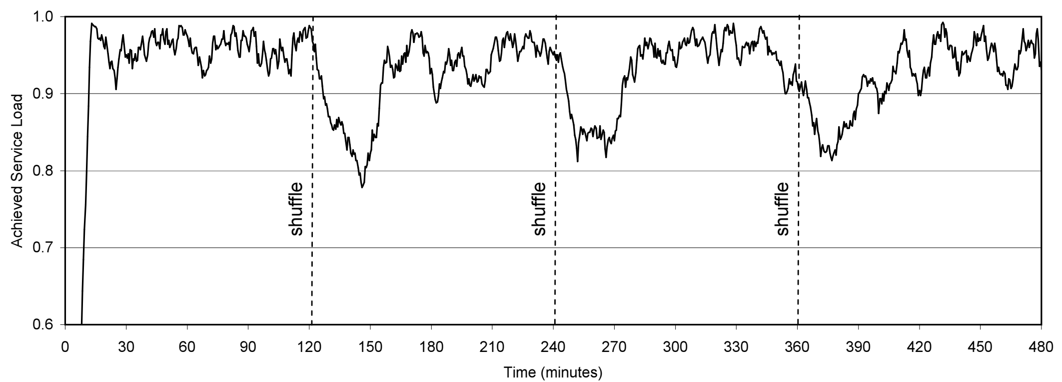
(b) Load Imbalance



(c) Replication Bandwidth



(d) Used Storage (% of Minimum Required)



(e) Achieved Load Over Time for 120 Minute Shuffle Period

Figure 6.5: Results of the changing popularity experiment

Configuration:	MinOne
Presentation properties	
Duration (sec)	758
Size (MB)	23
Data rate (Kbps)	272
Number of presentations	100
Experiment parameters	
Target load	1.0 (200,000Kbps)
Requests per hour	3,492
Total requests	27,936 (8 hours)
Shuffle period (minutes)	15, 30, 45, 60, 75, 90, 120
HammerHead Configuration	
Minimum replica count	1
Minimum count proportion	1.0
Cluster storage capacity (MB)	9,200

Table 6.3: Parameters used for the changing popularity experiment

and lowest average storage utilization is low, however, and, in this cluster configuration, was approximately equivalent to the storage capacity required to store two replicas.

Lower server performance for high shuffle frequencies is explained by the rate of change in the popularity distribution and the time taken for HammerHead to respond and adapt to each change. Figure 6.5 (e) shows the effect over time of each popularity shuffle on the achieved service load, for a shuffle period of 120 minutes. Once the popularity of the presentations is shuffled (at the times indicated by a broken line), the achieved service load begins to decrease, since the distribution of replicas no longer reflects client behaviour. When HammerHead's estimation of the relative popularity of each presentation begins to reflect the changing pattern of requests, replicas will be created, removed or relocated to reflect the change. The achieved service load is then further decreased, since inter-node replication streams reduce the bandwidth available for supplying streams to clients. Once the cluster has adapted to the new request pattern, the achieved service load begins to return to its original level as new client requests arrive. Once HammerHead as begun to adapt to a change in the relative popularity of the presentations, the time required to implement any required changes in the distribution of content depends on the size of the effected presentations and the data

rate of the inter-node replication streams. The creation or movement of a replica during this experiment took over twelve minutes – the duration of the presentations used. The impact of the redistribution of content was not fully reflected in the performance of the service for a further twelve minutes – the time taken for any client streams, started after the redistribution of replicas had been completed, to playback fully. Increasing the data rate for inter-node replication streams would reduce the time required to adapt to changes in demand but would increase the cluster bandwidth required by each replication stream.

Since popularity is likely to change gradually over days or weeks – rather than suddenly and frequently, as simulated in this experiment – the performance measured in this experiment is acceptable. Any change in the average duration of the streams supplied by the server, the data rate used for inter-node replication streams or the period over which client behaviour is estimated, will effect the responsiveness of the cluster to changes in demand. In practice, it would be necessary to find a suitable trade-off between the responsiveness of the cluster, the accuracy of estimations of demand and the proportion of server bandwidth used for creating and moving replicas. For example, it may be sufficient to evaluate the demand for each presentation, and the resulting assignment of replicas to cluster nodes, on an hourly or daily basis, rather than every five minutes, as was the case in each of the experiments described in this chapter.

6.2.4 Prototype Experiment 4: Reduced Storage Capacity

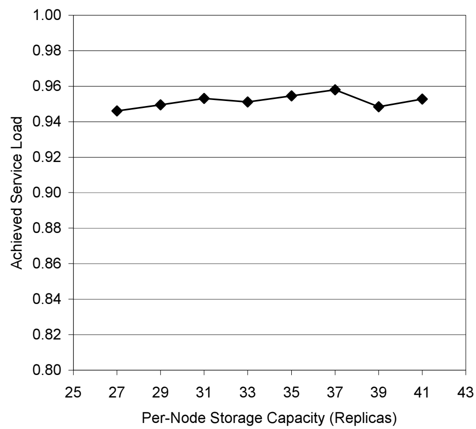
In each of the preceding three experiments, there was sufficient storage capacity on each node to store the entire multimedia archive (although, as has been shown by earlier results, only a proportion of the available space is required when using Dynamic RePacking). As a result, the Dynamic RePacking algorithm was not constrained by the storage capacity available on each node. This experiment evaluates the performance of the prototype server cluster when the per-node storage capacity is reduced and the Dynamic RePacking policy is constrained in its allocation of replicas to cluster nodes.

Configuration:	MinOne
Presentation properties	
Duration (sec)	758
Size (MB)	23
Data rate (Kbps)	272
Number of presentations	100
Experiment parameters	
Target load	1.0 (200,000Kbps)
Requests per hour	3,492
Total requests	13,968 (4 hours)
Shuffle period	no shuffle
HammerHead Configuration	
Minimum replica count	1
Minimum count proportion	1.0
Cluster storage capacity (MB)	2,484, 2,668, 2,852, 3,036, 3,220, 3,404, 3,588, 3,772
Cluster storage capacity (Replicas)	108, 116, 124, 132, 140, 148, 156, 164

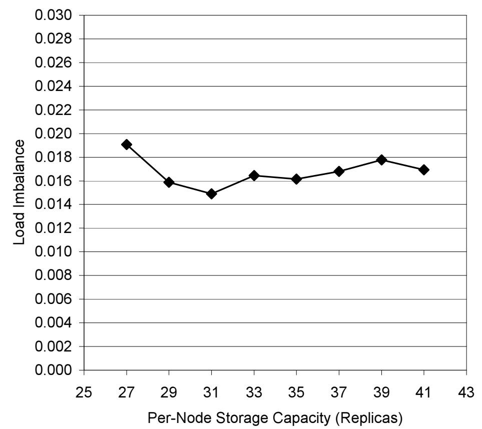
Table 6.4: Parameters used for the reduced storage capacity experiment

It will show that reducing the available storage capacity in this manner has little effect on the achieved service load.

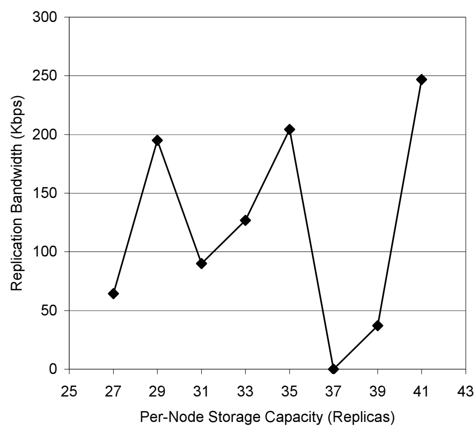
The *MinOne* Dynamic RePacking policy configuration is used again with the same set of one hundred presentations, the properties of which are listed in Table 6.4. The target cluster load was again 1.0. The minimum per-node storage capacity, in units of replicas, required to store a single replica of each of the 100 presentations was 25. This experiment examined the performance of the prototype server cluster when each node had sufficient capacity to store 27 replicas in the worst case (two more than the minimum) and 41 replicas in the best case (sixteen more than the minimum). Thus, the storage capacity of the cluster (and of each node, since each node has an identical storage capacity) ranged from 108% to 164% of the minimum required to store each presentation on exactly one node. In the most constrained case, Dynamic RePacking was forced to assign approximately the same number of replicas to each node. The experiment lasted four hours for each storage configuration. The results gathered over the first hour were excluded to allow the server time to stabilize. The parameters for the experiment are listed in Table 6.4 and the results are summarized in Figure 6.6.



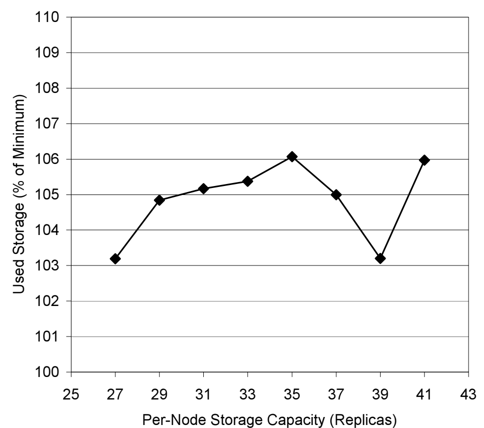
(a) Achieved Load as a Proportion of Maximum Load



(b) Load Imbalance



(c) Replication Bandwidth



(d) Used Storage (% of Minimum)

Figure 6.6: Results of the constrained storage capacity experiment

The results show that reducing the per-node storage capacity of the cluster has little effect on the achieved service load, which is comparable with that achieved when the Dynamic RePacking policy is unconstrained in its assignment of replicas to nodes. Although the results indicate some fluctuation in the average bandwidth used for the replication and movement of streams, the difference between the minimum and maximum average values is less than half of the bandwidth required by a single inter-node replication stream. The difference between the minimum and maximum storage capacity utilization is less than the capacity required to store three replicas.

6.2.5 Prototype Experiment 5: Heterogeneous Presentations

A multimedia server may be required a serve streams with different durations and bit rates and, as a result, store presentations with different file sizes. This experiment evaluates the impact of the use of presentations with randomly selected bit rates and durations on server performance. Although the results will show that the achieved service load varies when the characteristics of the most popular presentations are changed, the average decrease in achieved service load, when compared with the homogeneous case, is small.

The *MinOne* policy configuration used in previous experiments was used again. The storage capacity of the cluster was unconstrained. A set of one hundred presentations was created with bit-rates chosen randomly from a set of pre-defined stream profiles, ranging from 150Kbps to 459Kbps. The playback duration of each presentation was also selected at random, with a minimum duration of thirty seconds and a maximum duration of thirty minutes. The target cluster load was again 1.0.

Like the previous experiments, this experiment generated requests for each presentation with a probability determined by a Zipf distribution, resulting in highly skewed request patterns. Since the performance of the HammerHead cluster may be heavily influenced by the characteristics of the most popular presentation, the experiment was repeated eight times, with a different presentation ranked most popular each time. The results for each of these “shuffles” are compared with the results for the

Configuration:	MinOne
Presentation properties	
Duration (sec)	30 seconds – 30 minutes
Size (MB)	—
Data rate (Kbps)	150, 239, 368, 459
Number of presentations	100
Experiment parameters	
Target load	1.0 (200,000Kbps)
Requests per hour	generated dynamically
Experiment duration	4 hours
Shuffle period	no shuffle
HammerHead Configuration	
Minimum replica count	1
Minimum count proportion	1.0
Cluster storage capacity (MB)	9,200

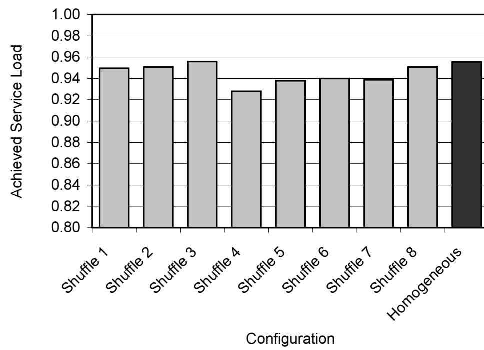
Table 6.5: Parameters used for the heterogeneous presentations experiment

MinOne policy obtained for Experiment 1 using a homogeneous set of presentations.

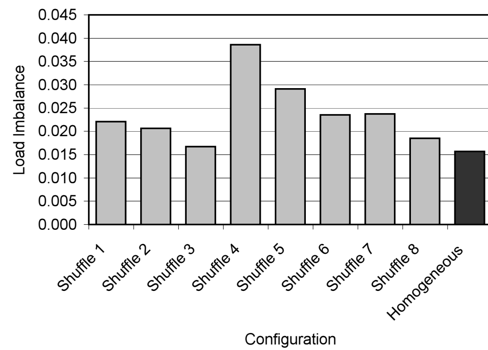
Each repetition of the experiment for a different popularity shuffle lasted four hours and the data from the first hour was excluded. For each of the repetitions, the mean request arrival rate required to achieve a target load of 1.0 was determined using equation 6.5. The parameters used to conduct the experiment are shown in Table 6.5 and the results obtained are summarized in Figure 6.7.

The achieved service load ranged from 0.928 to 0.956 for the best and worst shuffles, with an average of 0.944, compared with 0.955 for the homogeneous case. The average load imbalance ranged from 0.017 to 0.039 for the best and worst shuffles, with an average of 0.024, compared with 0.015 for the homogeneous case. The average number of replication streams started per hour in the heterogeneous configuration was 1.8, compared with 1.7 for the homogeneous configuration.

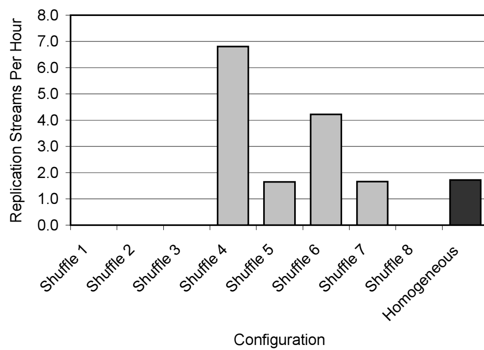
Since the presentations in a heterogeneous archive will have varying file sizes, the average number of replicas stored by the cluster, as a proportion of the minimum number of replicas (100) is presented in Figure 6.7(d), rather than the used storage capacity. The average storage utilization was approximately the same for both the heterogeneous (103.9%) and homogeneous (104.6%) cases.



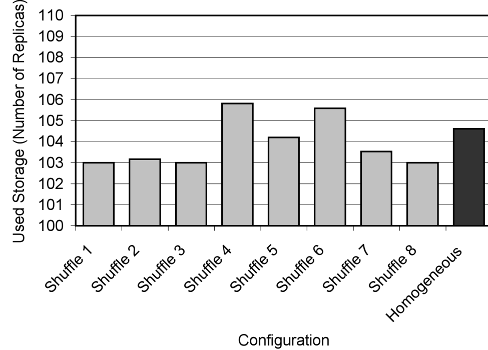
(a) Achieved Load as a Proportion of Maximum Load



(b) Load Imbalance



(c) Replication Streams Per Hour



(d) Used Storage (Number of Replicas)

Figure 6.7: Results of the heterogeneous presentations experiment

A more extensive evaluation of the performance of the Dynamic RePacking policy when supplying streams with varying bit-rates and durations, as well as clusters containing nodes with varying bandwidths and storage capacities, has been conducted using an event-driven simulation and is described in section 6.3.

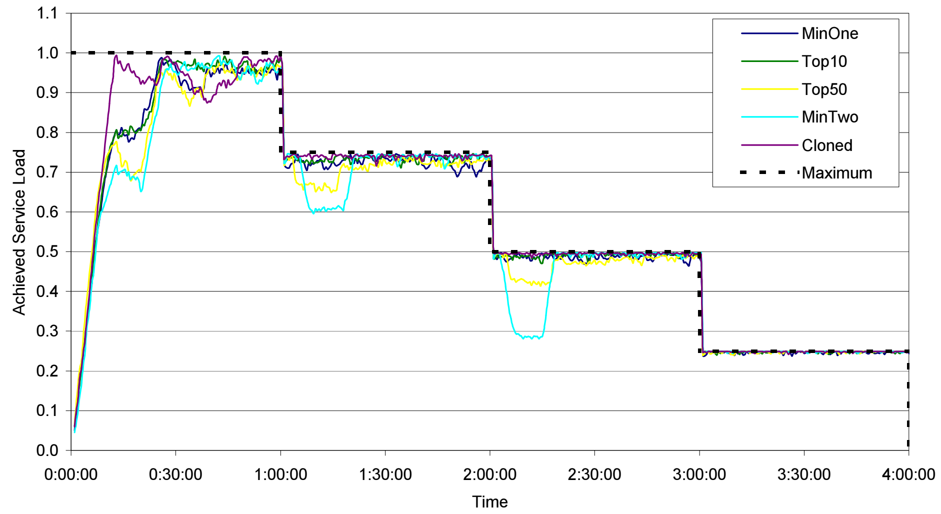
6.2.6 Prototype Experiment 6: Availability

The final experiment conducted on the prototype four-node cluster examines the performance of the cluster with respect to achieved load and harvest as nodes fail. The results demonstrate the trade-off between increasing the degree of replication – at the expense of increased storage utilization – and maintaining performance when failures occur.

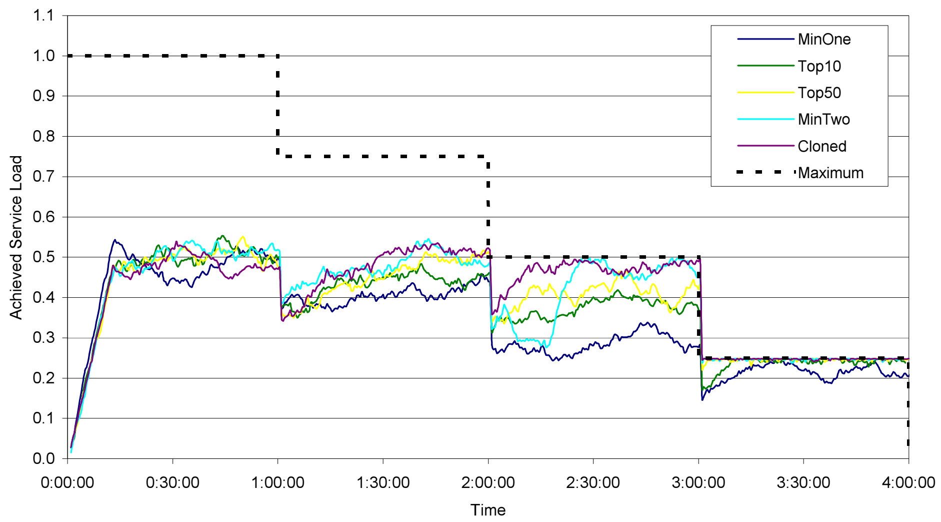
The five Dynamic RePacking policy configurations used in Experiment 1 (*MinOne*, *Top10*, *Top50*, *MinTwo* and *Cloned*) were again used in this experiment. For each of the five policy configurations, the four-node cluster was allowed one hour to stabilize before the first node failure occurred, after which one node was removed from the cluster every hour until each node had failed. The experiment was conducted for target cluster loads of 0.5 and 1.0, since the cluster load determines the impact of any reduction in harvest on the achieved yield and, as a result, achieved service load, as discussed in Chapter 2.

The same set of one hundred homogeneous presentations used in earlier experiments was used again. The parameters used for the configuration of the server and the generation of the simulated workload are listed in Table 6.6. The achieved service load (expressed as a proportion of the service capacity of the cluster before any failures occur) for target loads of 0.5 and 1.0 are plotted against time in Figures 6.8 (a) and (b) respectively. The broken lines in each of these figures represent the maximum service capacity of the cluster after each node failure. The harvest after each node failure for both target cluster loads is shown in Figure 6.9.

A cloned server cluster stores a replica of every presentation on every node and maintains 100% harvest despite any node failures. As a result, the cluster should be



(a) 100% Target Load

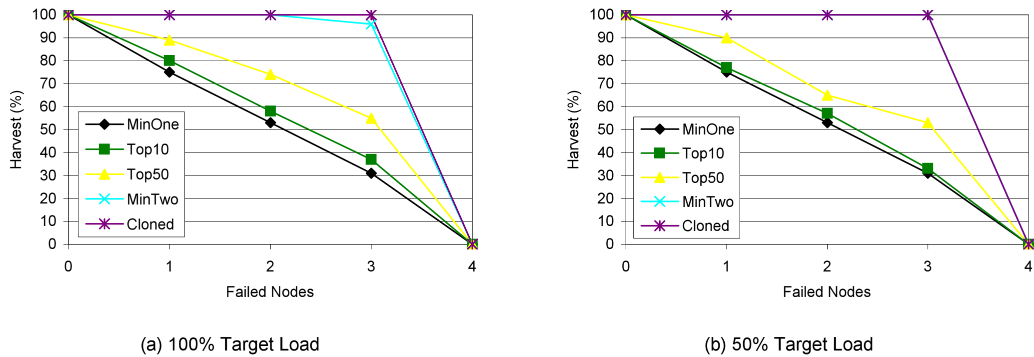


(b) 50% Target Load

Figure 6.8: Achieved service load after node failure

Configuration:	MinOne	Top10	Top50	MinTwo	Cloned
Presentation properties					
Duration (sec)	758				
Size (MB)	23				
Data rate (Kbps)	272				
Number of presentations	100				
Experiment parameters					
Target load	1.0 (200,000Kbps), 0.5 (100,000Kbps)				
Requests per hour	3,492, 1,746				
Total requests	13,968, 6,984				
Shuffle period	no shuffle				
HammerHead Configuration					
Minimum replica count	1	2	2	2	4
Minimum count proportion	1.0	0.1	0.5	1.0	1.0
Cluster storage capacity (MB)	9,200	9,200	9,200	9,200	9,200

Table 6.6: Parameters used for the availability experiment



(a) 100% Target Load

(b) 50% Target Load

Figure 6.9: Harvest after node failure

able to achieve a cluster load equal to the lower of the target load and the service capacity of the cluster, regardless of any node failure. This is the optimal performance achievable when nodes fail and is used as a baseline, against which the performance of the Dynamic RePacking policy can be measured under failure conditions.

The profile of the cloned configuration for a target cluster load of 1.0 is considered first. The achieved load using the *Cloned* policy builds from zero up to approximately 1.0 (200Mbps) over the playback duration of a single stream – until the target load is reached. At this point, since optimal load-balancing is achieved, each node approaches 100% utilization, as observed in Experiment 1. When the first node fails, one quarter of the active client streams fail and the achieved cluster load drops to 0.75 (150Mbps). Since there is no unused service capacity on any of the remaining nodes, the achieved load remains at 0.75 until the next failure occurs. Similar behaviour is observed for each failure until no nodes remain.

The observed behaviour of the cloned configuration for a target load of 0.5 (100Mbps) is different. When the first node failure occurs, the service capacity of the cluster drops to 75% of its original value. Since the target load is only 0.5, however, the remaining nodes have sufficient unused capacity to take over the workload of the failed node as new requests arrive and the achieved service load recovers towards the target load of 0.5 (100Mbps). Similar behaviour is observed after the second node failure. After the third node failure, however, there is no unused capacity on the single remaining node and the achieved service load remains at 0.25 (50Mbps) until the final node failure.

The behaviour and performance of each of the Dynamic RePacking configurations is now considered and compared with that of the cloned configuration. The behaviour of the *MinOne* and *Top10* configurations for a target load of 1.0 is comparable with that of the cloned configuration, with no spare capacity on remaining cluster nodes to take on the workload of failed nodes. In this case, the number of requests arriving for the remaining presentations, which were not lost when the failure occurred, is sufficient to fully utilize the bandwidth of the remaining nodes. Both the *Top50* and *MinTwo*

configurations, however, exhibit a significant drop in achieved service load after each failure, due to the use of cluster bandwidth for re-replication of presentations, up to their minimum required number of replicas. For example, in the case of the *MinTwo* configuration, each presentation is required to have at least two replicas. After each node failure, some of these replicas will be lost and Dynamic RePacking will recreate the lost replicas on the remaining nodes. This behaviour is not seen after the third node failure since, with only a single node remaining, the replacement replicas cannot be recreated.

Repeating the experiment with a target load of 0.5 demonstrates the effect of loss of harvest on the performance of the HammerHead cluster, when using Dynamic RePacking. After each node failure, those presentations with only a single replica on the failed node are lost, resulting in a decrease in harvest as shown in Figure 6.9(b). For each of the remaining presentations, with one or more replicas on the remaining nodes, fault-tolerant Dynamic RePacking will recreate the required minimum number of replicas for each presentation, as was observed earlier for a target load of 1.0. The nodes that remain after the first failure will have enough spare capacity to take on the workload of the failed node, as new client requests arrive. The loss in harvest resulting from the *MinOne*, *Top10* and *Top50* configurations, however, means that some requests will be rejected due to the unavailability of a subset of the presentations. The impact of any reduction in harvest, however, is reduced, since additional replicas exist for the most demanding presentations. For example, when using the *Top10* configuration, harvest decreases to 57% after two node failures but the achieved service load recovers to approximately 0.4 (the average over the thirty minutes before the final node failure occurs) or 80% of the target service load. In other words, a reduction in harvest of over 40% resulted in a reduction in yield of only 20%.

The reduction in harvest when a node fails depends on the location of replicas within the cluster, rather than on the cluster load, and the same trends are observed for both target loads, as shown in Figure 6.9. The *MinTwo* policy should maintain 100% harvest until the final failure occurs unless, after a failure, a subsequent failure occurs

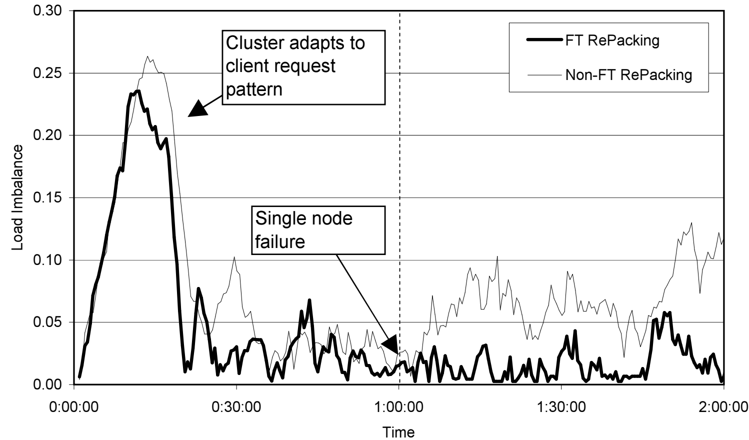


Figure 6.10: Fault-tolerant and non-fault-tolerant Dynamic RePacking

before the cluster has time to recover. The *MinTwo* configuration with a target load of 0.5 exhibits a small decrease in harvest (4 presentations become unavailable) after the third node failure. This is explained by the locking policy used by HammerHead to implement Dynamic RePacking. When moving a replica from a node A to another node B, when there is another replica of the same presentation on a node C, the replica on C may be chosen to supply the replication stream, allowing the replica on A to be removed as soon as any client streams have terminated. If node C fails before the replica has been created on B, all replicas of the presentation will be lost. A more rigourous approach to locking would lock the minimum number of replicas required by the presentation – regardless of whether those replicas are used to supply the replication stream – until the new replica has been created.

To demonstrate the advantage of fault-tolerant Dynamic RePacking over the basic Dynamic RePacking algorithm, the dynamic behaviour of a cluster with fault-tolerant Dynamic RePacking disabled was compared with that of a cluster with fault-tolerant Dynamic RePacking enabled. Both versions used the *MinOne* policy, creating only enough replicas to achieve load-balancing and maintain load-balancing after a single node failure. Dynamic RePacking was disabled when a node failure occurred, to prevent the cluster from recovering from the failure and allow the difference between the two algorithms to be demonstrated to greater effect. The parameters used to conduct the

experiment were the same as those shown in Table 6.6 but with a target load of 0.75. The change in load imbalance over time is shown in Figure 6.10 for both versions of the Dynamic RePacking policy, with the time at which the single node failure occurs highlighted by the broken line. It is clear from the figure that both versions achieve similar degrees of load-balancing up to the failure. After the failure, however, the non-fault-tolerant version exhibits a clear increase in load-imbalance, while the load-imbalance of the fault-tolerance version remains approximately the same. Fault-tolerant Dynamic RePacking resulted in the creation of two additional replicas over the non-fault-tolerant version in this experiment.

6.3 Dynamic RePacking Simulation

The time and resources required to perform the above experiments on the prototype HammerHead cluster prohibited further experimentation with larger cluster sizes, larger multimedia archives, a wider range of heterogeneous cluster configurations and varying client behaviour patterns. To examine the performance of the Dynamic RePacking policy under these conditions, an event driven simulation was used.

Client behaviour was simulated in the same manner as that used for experiments on the prototype cluster, with exponentially distributed inter-arrival times and presentation request probabilities determined by the Zipf distribution. The simulation operated at the resource reservation level, simulating the allocation of each node's bandwidth resources to client and replication streams and the allocation of storage resources to replicas. At this level, the behaviour of the simulation was designed to reflect that of the prototype cluster. Unless otherwise stated, each simulation experiment gathered results over a simulated seven-hour period (following a period during which the simulated cluster was allowed to stabilize) and each experiment was repeated ten times, with the average results presented. The experiments conducted were designed to evaluate:

1. How well the results from the event-driven simulation match those obtained from

the prototype HammerHead server cluster.

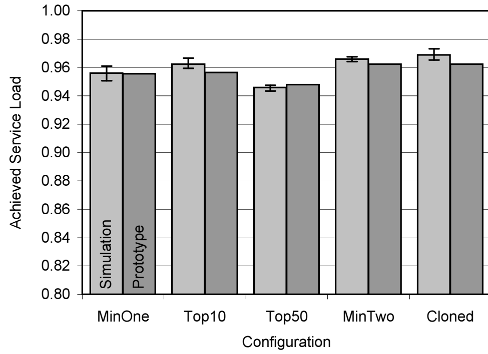
2. The effect of increasing the number of presentations in an archive.
3. The effect of increasing the number of nodes in a cluster.
4. The effect of varying the parameter θ to vary the skew in the popularity of the presentations in an archive.
5. The performance of a number of heterogeneous cluster configurations.

6.3.1 Simulation Experiment 1: Comparison of Simulation and Prototype Results

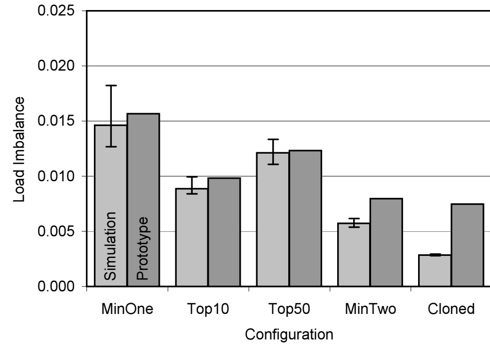
To compare the results obtained using simulation with those obtained for the prototype cluster, the first five experiments described in Section 6.2 were repeated using the event-driven simulation. In each case, the parameters for the experiment were the same for both the prototype experiments and the simulation. Unlike the prototype experiments, which used the same initial random assignment of replicas to nodes, the simulation generated a new random placement for each repetition. The results obtained from the simulation experiments are shown in Figures 6.11 to 6.14, alongside the results from the prototype experiments.

In the case of the experiment comparing different Dynamic RePacking policy configurations, the maximum and minimum values obtained from the repetitions of the simulation experiment are indicated. With respect to achieved load, the simulation results are within 0.7% of the results obtained from the HammerHead prototype.

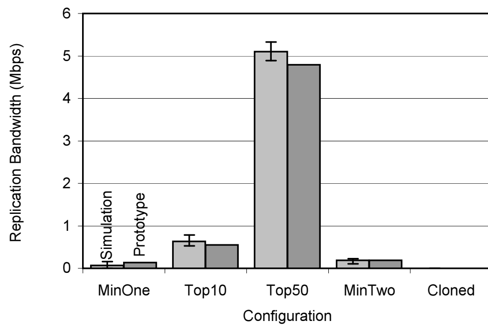
It is worth noting the difference between the results obtained from the HammerHead prototype and the simulation for the cloned cluster configuration, since they are not influenced by the Dynamic RePacking policy. There is a difference of 0.008 between the minimum and maximum achieved service load in the simulation of this configuration and a negligible margin of error in the load imbalance. The load imbalance of the HammerHead prototype is, however, significantly higher. This difference results from a lack of up-to-date knowledge in the HammerHead prototype of the precise resource



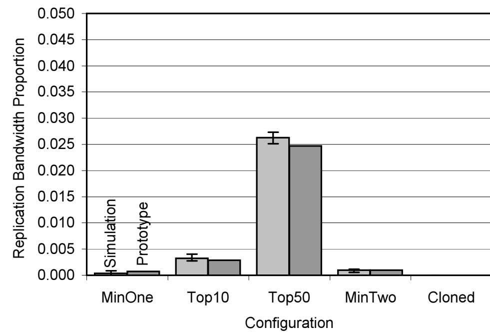
(a) Achieved Load as a Proportion of Maximum Load



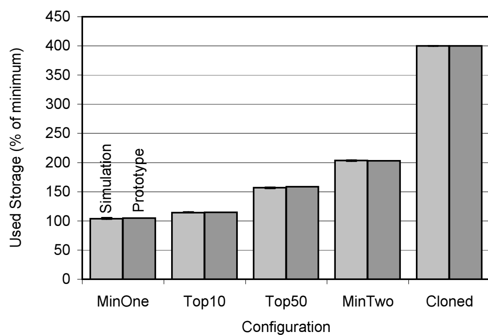
(b) Load Imbalance



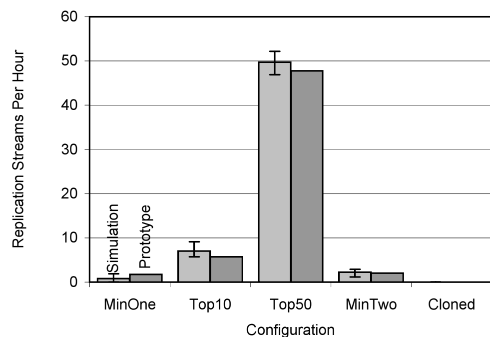
(c) Replication Bandwidth



(d) Replication Bandwidth as a Proportion of Total Cluster Bandwidth



(e) Used Storage (% of Minimum Required)



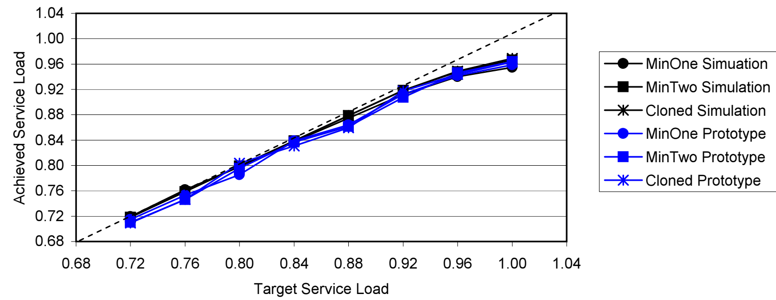
(f) Replication Streams Started per Hour

Figure 6.11: Comparison of simulation and prototype results for the replication policy experiment

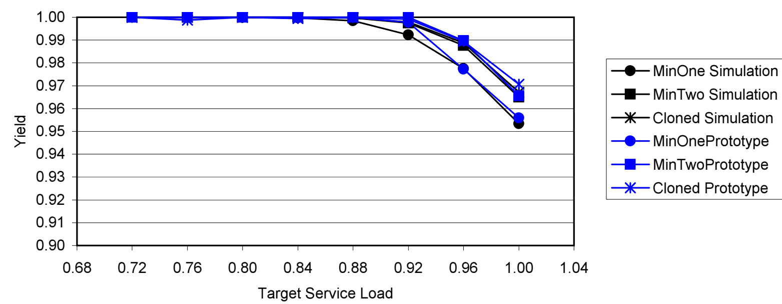
utilization of each node in the cluster, due to the latency between the start of a stream on a cluster node and the arrival of the STREAM_START event at the `HammerServer` layer. In addition, there is a delay between the redirection of a client request and the eventual start of the associated stream. In other words, a `HammerServer` instance will redirect some clients to nodes which are fully utilized by the time the redirected client request reaches the node. Similarly, a `HammerServer` instance will reject some requests when resources are available at a suitable Windows Media Services instance but before the `HammerServer` instance learns of the increased availability. It is possible to quantify those requests that are accepted by `HammerServer` instances, but are later rejected by Windows Media Services, by examining the logs created by the workload generator. For example, in the case of the *MinOne* policy, 81.8% of the requests rejected were rejected by a Windows Media Services instance, rather than by a `HammerServer` instance. A similar proportion (78.6%) were rejected by Windows Media Services in the case of the *MinTwo* configuration. This figure rises to 99.0% for the cloned configuration. As discussed in Section 5.4.1, `HammerSource` plug-ins batch HammerHead events together, sending batched events to the `HammerServer` layer every two seconds. This delay could be reduced to increase the consistency of the `HammerServer` layer's knowledge of the cluster state, at the expense of an increased group communication overhead.

The results of the simulation experiment for different target cluster loads are compared in Figure 6.12 with those obtained from the HammerHead prototype. The simulation exhibits similar trends to the prototype with respect to achieved load, yield and load imbalance.

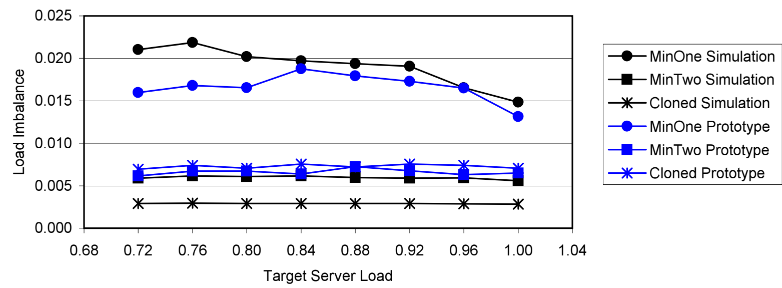
Figure 6.13 compares the simulation and prototype results for varying popularity "shuffle" frequencies. The shuffle frequencies used represent extreme cases that require frequent redistribution of replicas among cluster nodes. With respect to achieved load, the simulation results differ from those obtained from the prototype experiments by less than 2%, with both sets of results exhibiting the same trends. The prototype results are within the range of results obtained from the simulation.



(a) Achieved Load as a Proportion of Maximum Load

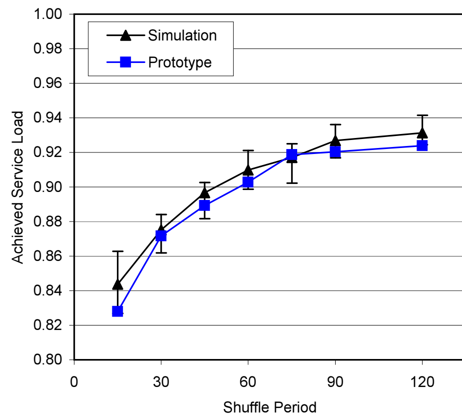


(b) Yield

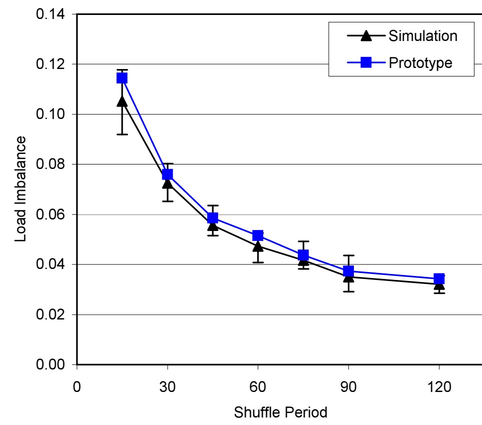


(c) Load Imbalance

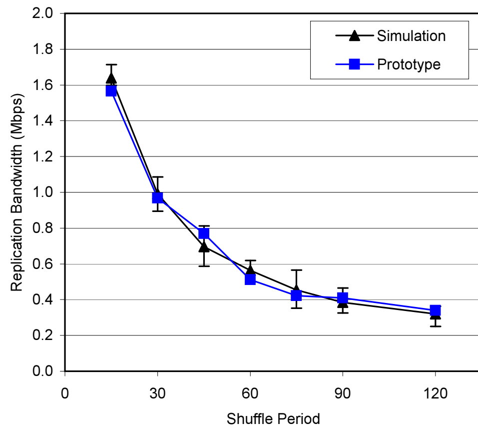
Figure 6.12: Comparison of simulation and prototype results for the target cluster load experiment



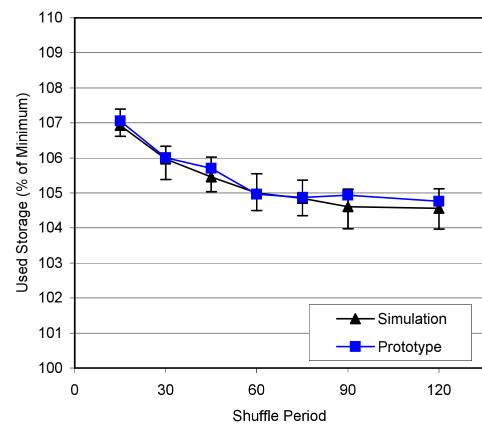
(a) Achieved Load as a Proportion of Maximum Load



(b) Load Imbalance

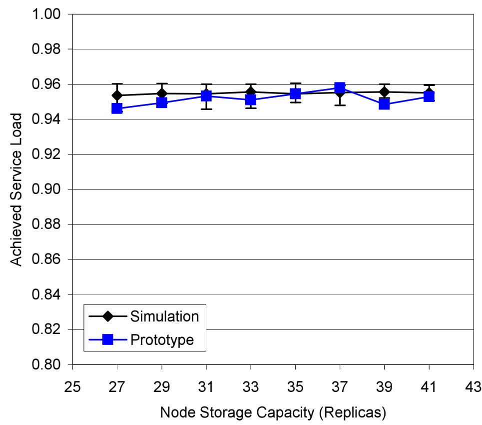


(c) Replication Bandwidth

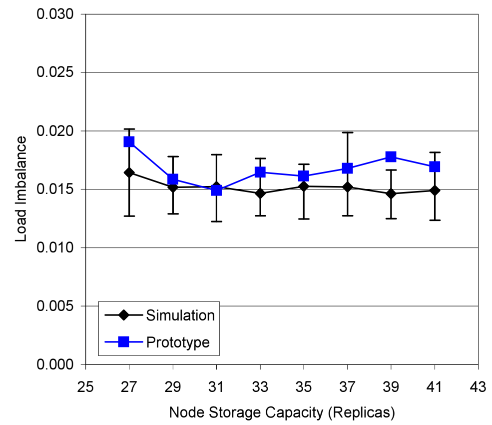


(d) Used Storage (% of Minimum Required)

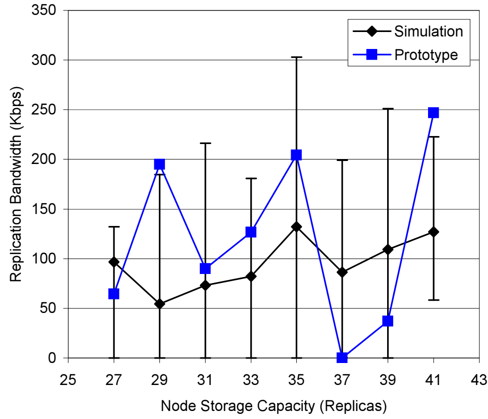
Figure 6.13: Comparison of simulation and prototype results for the changing presentation popularity experiment



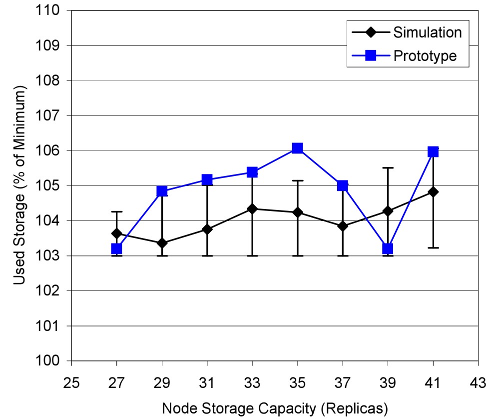
(a) Achieved Load as a Proportion of Maximum Load



(b) Load Imbalance



(c) Replication Bandwidth



(d) Used Storage (% of Minimum)

Figure 6.14: Comparison of simulation and prototype results for a cluster with reduced storage capacity

Configuration:	MinOne
Presentation properties	
Duration (sec)	758
Size (MB)	23
Data rate (Kbps)	272
Number of presentations	500, 1,000, 1,500, . . . , 5,000
Experiment parameters	
Target load	1.0 (200,000Kbps)
Simulated time	10 hours (first 3 hours excluded from results)
Shuffle period	no shuffle
HammerHead Configuration	
Number of cluster nodes	4
Minimum replica count	1
Minimum count proportion	1.0
Cluster storage capacity	unconstrained

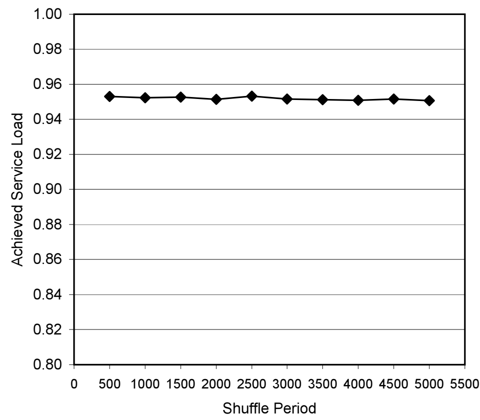
Table 6.7: Parameters used to evaluate the effect of increasing the archive aize

Finally, the results obtained from the prototype HammerHead cluster with reduced storage capacity are compared with those obtained from the simulation in Figure 6.14. Although there is some deviation in the results, the deviation is small and the results are still comparable. It should be noted that each data point for the prototype represents the average performance over a single three-hour period, while the simulation results represent the average over a seven-hour period and the simulation was repeated ten times. Despite these differences, both sets of results exhibit the same characteristics.

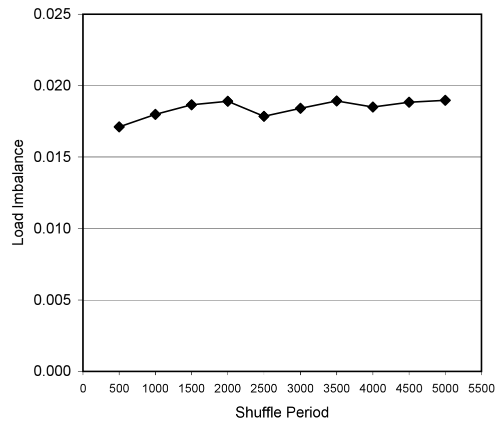
6.3.2 Simulation Experiment 2: Increasing Archive Size

The event-driven simulation was used to evaluate the effect on performance of increasing the number of presentations in the multimedia archive. The parameters for the simulation experiment are shown in Table 6.7. The storage capacity of each node was unconstrained. The results of the experiment, shown in Figure 6.15, indicate near-uninform performance with respect to achieved load.

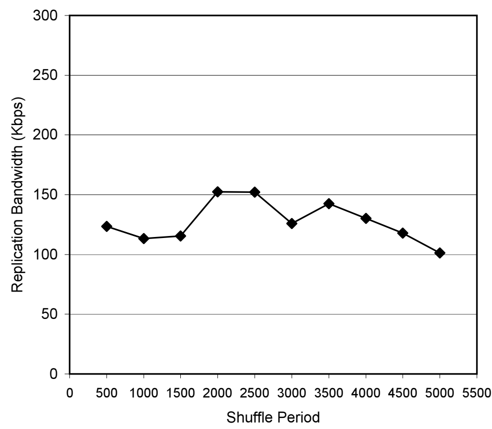
The minor increase in load imbalance with increasing numbers of presentations is explained by the reduction in the proportion of requests for presentations of the same



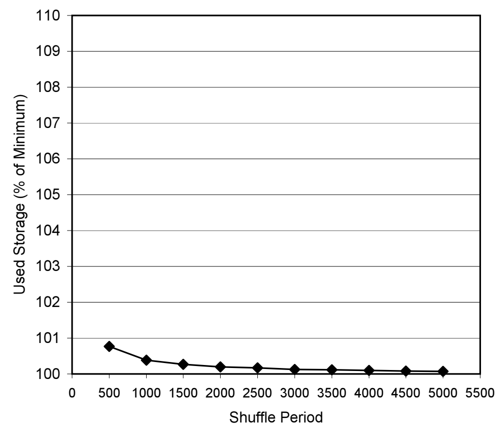
(a) Achieved Load as a Proportion of Maximum Load



(b) Load Imbalance



(c) Replication Bandwidth



(d) Used Storage (% of Minimum Required)

Figure 6.15: Results of the archive size experiment

rank in archives of different sizes, as determined by the Zipf distribution. For example, the presentation ranked fifth by popularity in an archive with 100 presentations will receive over 75% more requests than the presentation ranked fifth in an archive with 5000 presentations. This has the effect of reducing the accuracy of the prediction of demand for each presentation. The bandwidth required by inter-node replication streams is almost independent of the number of presentations in the archive, with the difference between the maximum and minimum average values equivalent to less than 10% of the bandwidth of a single replication stream.

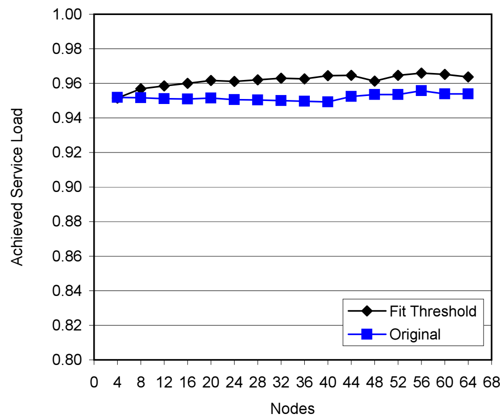
Storage utilization – when expressed as a proportion of the minimum storage capacity required to store a single replica of each presentation – decreases, since the number of presentations with more than one replica created for load balancing is independent of the number of presentations in the archive. This result further demonstrates the benefit of Dynamic RePacking over complete replication of the archive on every cluster node and, in particular, how the benefit increases with the archive size.

6.3.3 Simulation Experiment 3: Increasing Cluster Size

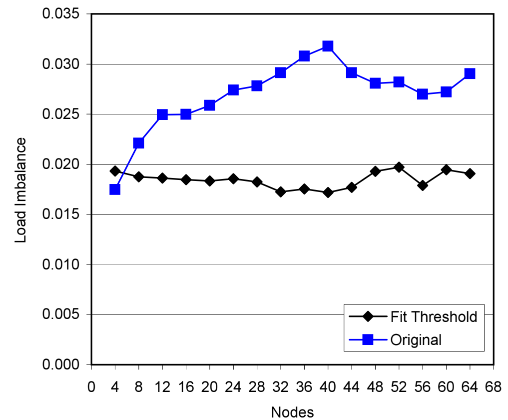
This experiment examines the effect on the performance of Dynamic RePacking of increasing the number of nodes in a cluster and shows that the service load achieved for smaller clusters can be maintained as the cluster size increases.

Clusters containing from four to 64 nodes have been evaluated, and the characteristics of each node were the same as those used in the prototype and simulation experiments described earlier. The number of presentations was fixed at one thousand, which has the effect of storing fewer replicas on each cluster node as the number of nodes increases.

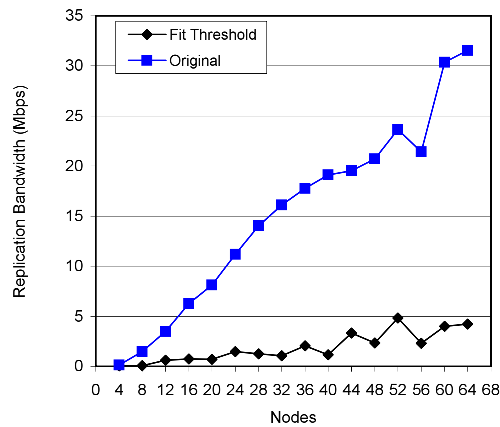
The Dynamic RePacking policy creates more replicas for the most demanding presentations as the number of nodes increases, to facilitate an even distribution of the expected client workload across each node. For large numbers of nodes, small changes in the relative demand for each presentation cause the relocation of an increasing number of these additional replicas, since the demand for each presentation is distributed



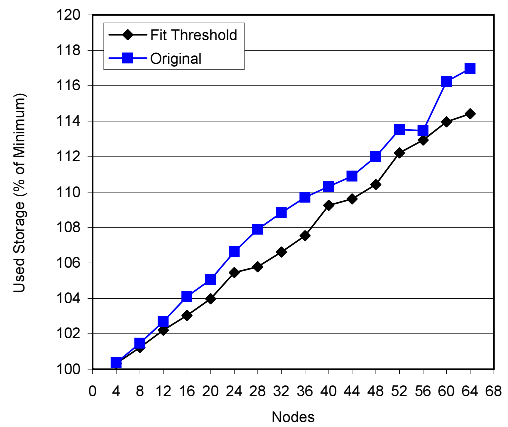
(a) Achieved Load as a Proportion of Maximum Load



(b) Load Imbalance



(c) Replication Bandwidth



(d) Used Storage (% of Minimum Required)

Figure 6.16: Results of the cluster size experiment

Configuration:	MinOne
Presentation properties	
Duration (sec)	758
Size (MB)	23
Data rate (Kbps)	272
Number of presentations	1,000
Experiment parameters	
Target load	1.0
Simulated time	10 hours (first 3 hours excluded from results)
Shuffle period	no shuffle
HammerHead Configuration	
Number of cluster nodes	4, 8, 12, ..., 64
Minimum replica count	1
Minimum count proportion	1.0
Cluster storage capacity	unconstrained

Table 6.8: Parameters used to evaluate the effect of increasing the cluster size

among many nodes. To counteract this effect, the modification to Dynamic RePacking described in Section 4.2.5 is used to determine how well the current distribution of replicas fits the revised expected demand. If the aggregate demand of the presentations that remain unpacked, after applying Dynamic RePacking without the ability to create new replicas, exceeds a specified "fit threshold", then Dynamic RePacking is performed. Otherwise, the current distribution of replicas is left unaltered. The "fit threshold" used for the experiment was 0.02, causing Dynamic RePacking to be performed if the unallocated demand was greater than 2% of the demand for the streaming service. The choice of fit threshold was motivated by the requirement to allow the cluster to adapt to changes in the relative demand for presentations, without performing excessive replication.

The effect of increasing the cluster size, both with and without the "fit threshold" modification, has been evaluated. The parameters for the experiment are shown in Table 6.8 and the results are presented in Figure 6.16.

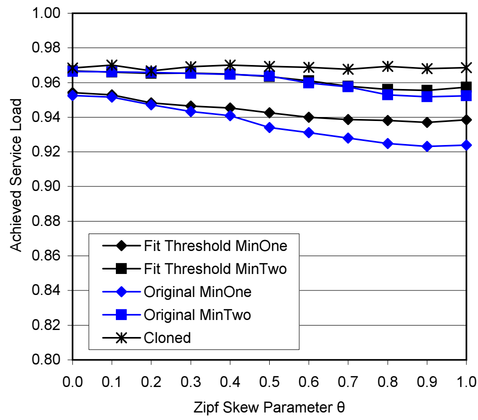
An increasing number of replicas of the most demanding presentations results in an increase in the achieved load. Without the "fit threshold", this increase is counteracted by an increase in the cluster bandwidth required to perform inter-node replication.

The used storage capacity – expressed as a proportion of the minimum capacity required to store one replica of each presentation – increases as nodes are added to the cluster, since more replicas of the most demanding presentations are created to achieve load balancing. The storage capacity used by the simulated *MinOne* policy configuration in a cluster containing 64 nodes is less than 1.8% of that required by a cloned server cluster. Additional simulation experiments showed the average achieved load for a cloned 64 node cluster to be 0.990, compared with 0.964 for the *MinOne* policy configuration. By creating at least two replicas of each presentation in a 64 node cluster, the achieved load increases to 0.980, while using approximately 3.5% of the storage capacity required by the cloned cluster.

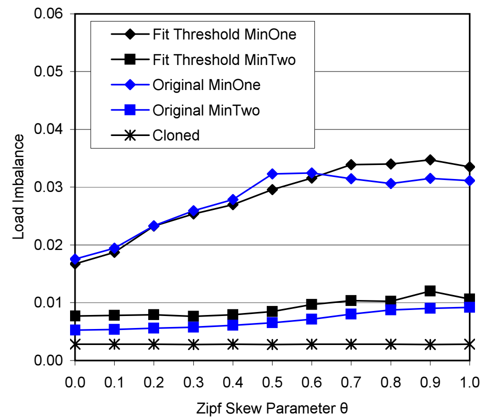
6.3.4 Simulation Experiment 4: Popularity "Skew" Parameter θ

All of the simulation and prototype experiments described so far use a Zipf distribution with parameter $\theta = 0$, as discussed in Section 2.2.1, to simulate a skewed distribution of the relative popularity for each presentation. This simulation experiment evaluates the effect on performance of increasing the value of θ towards a uniform popularity distribution with $\theta = 1.0$. The parameters for the experiment are shown in Table 6.9 and the results are shown in Figure 6.17. Both the *MinOne* and *MinTwo* Dynamic RePacking configurations were simulated and a cloned configuration was simulated for comparison.

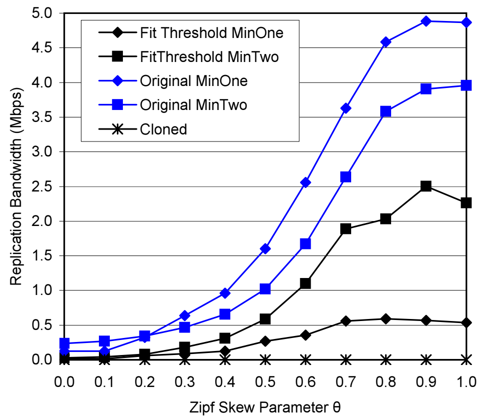
The results show an increase in load imbalance as θ increases towards 1.0 – which represents a uniform distribution of popularity – resulting in a decrease in achieved service load. The bandwidth required by inter-node replication streams increases as the relative demand for each presentation approaches $1/K$, where K is the number of presentations in the archive. This increase results from frequent changes in the estimated relative demand for each presentation and corresponding changes in the assignment of replicas to nodes. To counteract this effect, the experiment was repeated using the "Fit Threshold" modification described in Section 4.2.5, again with a threshold of 0.02. This had the effect of reducing the rate of replica movement and increasing the



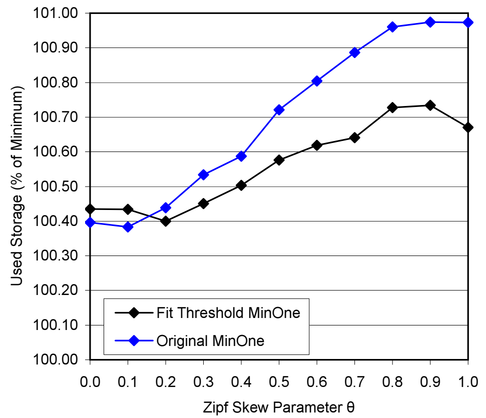
(a) Achieved Load as a Proportion of Maximum Load



(b) Load Imbalance



(c) Replication Bandwidth



(d) Used Storage (% of Minimum Required)

Figure 6.17: Results of the popularity skew experiment

Configuration:	MinOne	MinTwo	Cloned
Presentation properties			
Duration (sec)	758		
Size (MB)	23		
Data rate (Kbps)	272		
Number of presentations	1,000		
Experiment parameters			
Target load	1.0 (200,000Kbps)		
Simulated time	10 hours (first 3 hours excluded from results)		
Shuffle period	no shuffle		
Skew parameter θ	0.0, 0.1, ..., 1.0		
HammerHead Configuration			
Number of cluster nodes	1	2	4
Minimum replica count	1	2	4
Minimum count proportion	1.0	1.0	1.0
Cluster storage capacity (MB)	unconstrained	unconstrained	unconstrained

Table 6.9: Parameters used to evaluate the effect of changing the popularity skew parameter, θ

achieved service load. The effect on load imbalance was small, however, since the use of the fit threshold does not improve the accuracy of demand estimation. In practice, if the distribution of popularity was known to be near-uniform, the period over which the relative demand for each presentation is estimated could be increased and greater weight could be applied to older measurements.

6.3.5 Simulation Experiment 5: Heterogeneous Cluster Configurations

The final simulation experiment evaluates the effect on performance when Dynamic RePacking is applied to clusters with varying bandwidth and storage capacities and to archives containing presentations with varying bit-rates and playback durations. The parameters for the simulation experiment are shown in Table 6.10. A number of different heterogeneous configurations have been evaluated and the parameters for each configuration are shown in Tables 6.11 and 6.12. The results demonstrate that Dynamic RePacking can be applied to heterogeneous server configurations with only a

Configuration:	MinOne
Presentation Properties	
Number of presentations	1,000
Experiment Parameters	
Target load	1.0
Simulated time	10 hours (first 3 hours excluded from results)
Shuffle period	no shuffle
HammerHead Configuration	
Number of cluster nodes	4
Minimum replica count	1
Minimum count proportion	1.0

Table 6.10: Parameters used in heterogeneous cluster simulation experiments

small reduction in performance.

When unpacking replicas, to make storage capacity available for replicas of more demanding presentations, as described in Section 4.2.3, the selection of replicas to unpack is dependent on the estimated demand of each presentation. Since the accuracy of this estimation for unpopular presentations will be low, their relative ranking may change significantly between invocations of the Dynamic RePacking algorithm. This frequent change in the ranking of presentations may cause different replicas to be unpacked during each Dynamic RePacking invocation, increasing the cluster resources required by inter-node replication streams. To counteract this effect, the "fit threshold" modification to Dynamic RePacking, described in Section 4.2.5 is used for each of the simulation experiments described below, with a fit threshold of 0.02.

In the first heterogeneous configuration, labelled A, each cluster node is identical and the bit-rates and playback durations of the presentations in the archive are selected at random. The bit-rate of each presentation is between 100Kbps and 500Kbps and the duration of each presentation is between 30 seconds and 2 hours. The size of each presentation is a function of its random bit-rate and duration. The experiment was repeated twenty times for different randomly generated archives of 1,000 presentations and the average results are shown in Table 6.13. The parameters and results

Node	1	2	3	4
A. Presentation Bit-rate and Duration				
Node bandwidth	50,000			
Node storage	402,000			
Presentation bit-rate	random (100Kbps – 500Kbps)			
Presentation duration	random (30 seconds – 2 hours)			
B. Node Storage Capacity				
Node bandwidth	50,000	50,000	50,000	50,000
Node storage	2,875	2,875	23,000	23,000
Presentation bit-rate	272Kbps			
Presentation duration	758 seconds			
X. Equivalent Homogeneous Configuration				
Node bandwidth	50,000			
Node storage	23,000			
Presentation bit-rate	272Kbps			
Presentation duration	758 seconds			

Table 6.11: Heterogeneous cluster configurations A and B, and equivalent homogeneous configuration X

Node	1	2	3	4
C. Node Bandwidth				
Node bandwidth	50,000	50,000	500,000	500,000
Node storage	23,000	23,000	23,000	23,000
Presentation bit-rate	272Kbps			
Presentation duration	758 seconds			
D. Node Bandwidth and Storage Capacity				
Node bandwidth	50,000	50,000	500,000	500,000
Node storage	2,875	2,875	23,000	23,000
Presentation bit-rate	272Kbps			
Presentation duration	758 seconds			
E. Bandwidth, Capacity, Bit-Rate and Duration				
Node bandwidth	50,000	50,000	500,000	500,000
Node storage	2,875	2,875	2,875	2,875
Presentation bit-rate	random (100Kbps – 500Kbps)			
Presentation duration	random (30 seconds – 2 hours)			
Y. Equivalent Homogeneous Configuration				
Node bandwidth	275,000			
Node storage	23,000			
Presentation bit-rate	272Kbps			
Presentation duration	758 seconds			

Table 6.12: Heterogeneous cluster configurations C, D and E, and equivalent homogeneous configuration Y

Cluster service capacity = 200,000Mbps				Cluster service capacity = 1,100,000Mbps				
Configuration	A	B	X	**Configuration**	C	D	E	Y
Achieved load	0.943	0.951	0.952	Achieved load	0.979	0.962	0.967	0.983
Yield	0.949	0.950	0.953	Yield	0.980	0.962	0.975	0.983
Load imbalance	0.025	0.021	0.018	Load imbalance	0.007	0.020	0.028	0.005
Replication rate	0.47	0.30	0.20	Replication rate	0.02	0.59	0.26	0.04

Table 6.13: Results of the heterogeneous cluster experiments

for an equivalent four-node cluster configuration, but an archive containing 1,000 homogeneous presentations, are also provided for comparison in Tables 6.11 and 6.13 respectively, and are labelled *X*.

The results show that the achieved load, when supplying streams of heterogeneous presentations, is only 1% less than that for the homogeneous archive. (This difference is approximately the same as that obtained using a heterogeneous archive with the prototype HammerHead cluster in Prototype Experiment 5.)

In the second heterogeneous cluster configuration, labelled *B*, the presentations in the archive are homogeneous and the cluster is constructed from nodes with identical bandwidths but varying storage capacities. This has the effect of forcing the Dynamic RePacking policy to unpack less demanding presentations, replacing them with more demanding ones, as described in Section 4.2.3. The unpacked or *victim* replicas are placed on any node with sufficient storage capacity after the basic Dynamic RePacking algorithm has completed. The results of this configuration are again compared with those of the equivalent homogeneous configuration, labelled *X*, which has the same cluster service capacity but the storage capacity on each node is unconstrained.

Although there is a small increase in load imbalance and in the rate of movement of replicas, the performance achieved by the heterogeneous configuration, with respect to achieved load, is almost the same as that for the equivalent homogeneous configuration, shown in the column labelled *X*.

In the heterogeneous configuration labelled *C* (Table 6.12), an archive containing homogeneous presentations is again used and the cluster is constructed from nodes with

identical storage capacities but varying network bandwidths. Specifically, the bandwidth of the third and fourth nodes is increased by a factor of ten, relative to that of the first and second nodes. The average results for this configuration are again shown in Table 6.13. The parameters and results for an equivalent homogeneous four-node cluster configuration, with the same aggregate cluster bandwidth as the heterogeneous configuration (1,100,000Kbps) but using nodes with identical bandwidths (275,000Kbps), are shown for comparison and are labelled *Y*. The achieved service load was approximately 0.4% less than the equivalent homogeneous configuration.

The configuration labelled *D* (Table 6.12) combines the heterogeneous storage and bandwidth characteristics of configurations *B* and *C* respectively. The achieved service load was approximately 2% less than that for the equivalent homogeneous configuration, labelled *Y*.

The final configuration, labelled *E* (Table 6.12), combines the heterogeneous presentation and node characteristics of configurations *A* and *D* respectively, serving streams of heterogeneous presentations from a cluster of nodes with heterogeneous network bandwidths and storage capacities. The decrease in achieved service load, compared with the equivalent homogeneous configuration, labelled *Y*, was approximately 1.6%.

6.4 Summary

The cost – both in terms of time and equipment – of the real-time evaluation of the prototype HammerHead server cluster was high, with the experiments described in Section 6.2 requiring over twelve days to complete, assuming they could be executed continuously, without interruption. To further evaluate the performance of the Dynamic RePacking policy, a compressed-time event-driven simulation was used. A number of the experiments performed on the prototype HammerHead cluster were also simulated to verify the simulation. Although simulation was performed at the resource reservation level, and did not simulate the Ensemble group communication system or any network

characteristics beyond the streaming bandwidth of each node, the results obtained show that the event-driven simulation may be used to predict the performance and behaviour of the prototype cluster.

The experiments with the prototype cluster were used to evaluate the performance of the HammerHead architecture and the Dynamic RePacking replication policy, for a four-node cluster with a multimedia archive containing one hundred presentations. The impact of different Dynamic RePacking policy configurations, different target service loads, changing client request patterns, constrained per-node storage capacities and archives containing presentations with different streaming characteristics were all evaluated. The performance of Dynamic RePacking was shown to approach that of the cloned cluster configuration, with a significant reduction (almost 75% in the case of the *MinOne* policy configuration) in storage capacity utilization, demonstrating that selective replication of content is a viable approach to the implementation of an on-demand multimedia streaming service based on the server cluster model.

An event-driven simulation was used to predict the performance of a HammerHead cluster with larger archives and larger numbers of cluster nodes. In both cases, the achieved service load was almost uniform. In particular, the simulation showed that larger clusters can approach the performance of a cloned server cluster while using significantly less storage capacity. For example, a 64-node cluster using the *MinOne* Dynamic RePacking configuration requires less than 1.8% of the storage capacity of a cloned server cluster, with a reduction in achieved service load of only 2.6%.

A number of heterogeneous configurations – with clusters constructed from nodes with varying network bandwidths and storage capacities and archives containing presentations with randomly selected bit-rates and durations – have also been simulated and the results predict that these configurations perform almost as well as the equivalent homogeneous configurations.

The effect of node failures on cluster performance was demonstrated by experiments performed on the prototype HammerHead cluster. For a target service load of 1.0, with the exception of short recovery periods, the performance of HammerHead

using the Dynamic RePacking policy was close to that of a cloned configuration and the impact of any loss in harvest was counteracted by the absence of spare service capacity to take on the workload of failed nodes. For lower target loads, any decrease in harvest will reduce yield of the service, since some presentations will be unavailable. The impact of node failures can be reduced by creating additional replicas of important presentations or by creating a minimum number of replicas for each presentation. The use of fault-tolerant Dynamic RePacking has been shown to maintain load-balancing in the presence of node failures or when nodes are otherwise removed from the cluster.

Chapter 7

Conclusions and Future Work

This thesis has examined the provision of on-demand multimedia streaming services using clusters of commodity PCs. This examination takes place in the context of recent experiences of the provision of large-scale, highly-available services over the internet, of which the Google web search engine is perhaps one of the better known examples. Services such as these must scale incrementally, to reflect changes in the demand for the service or to increase redundancy. High-availability is also required, with performance degrading gracefully with successive component failures. Availability can be increased by providing redundant service capacity and storing redundant data. High performance is achieved by exploiting the inherent parallelism in the workload – client requests can be distributed among cluster nodes, with each node independently managing a disjoint subset of requests. Maximizing the performance of the cluster, however, requires the workload to be evenly distributed among the cluster's nodes. When complete replication of the data required by a service is impractical, either because of the storage equipment cost, management cost, environmental issues or the time required to replicate data on a new node, the data can be partitioned and replicated to achieve load balancing or increase the availability of individual data items.

Of the existing multimedia content distribution techniques, *dynamic replication* is the most suitable for distributing content in a manner that satisfies the requirements of an on-demand multimedia streaming service based on a cluster of commodity PCs.

Dynamic replication policies assign a subset of the presentations in a multimedia archive to each node in a cluster, such that the estimated demand for each node is the same, creating additional replicas of a subset of the presentations to facilitate load-balancing, satisfy high client demand or increase the availability of individual presentations.

The *HammerHead* architecture described in this thesis is the first multimedia server cluster architecture to combine the use of group communication with the implementation of a dynamic replication policy. This thesis also presents what is believed to be the first *implementation* of a multimedia server cluster using dynamic replication, to appear in the literature. HammerHead uses the facilities provided by a commodity multimedia streaming server and adds a cluster-aware layer, allowing the service provided by the commodity server to scale beyond a single node, without requiring complete replication of large multimedia archives. The cluster-aware layer maintains the aggregated state of the commodity streaming server instance on each cluster node. This aggregated state is used to redirect incoming client RTSP requests to suitable nodes, in a manner that takes the location of the replicas of the requested presentation, the state of the replicas and the current activity of each node into account. The aggregated state is also used to implement the dynamic content replication policy. To allow the task of redirecting client requests to be shared among the cluster's nodes, and allow the implementation of the dynamic replication policy to withstand multiple node failures, the aggregated cluster state is replicated on some or all of the cluster's nodes.

A new dynamic replication policy, called *Dynamic RePacking*, has been developed for use in the prototype HammerHead server cluster, since none of the existing policies described in the literature satisfied all of the requirements of the HammerHead architecture. The policy is a significant development of the existing MMPacking policy, capable of handling presentations with heterogeneous stream characteristics and clusters of heterogeneous nodes. The cost of adapting to changes in client behaviour is reduced by attempting to repack replicas on nodes where they are already stored. Dynamic RePacking initially creates a near-minimal number of additional replicas of a small number of demanding presentations, to allow load-balancing to be achieved.

A minimum required number of replicas can then be assigned to individual presentations to meet specific application requirements, such as increased availability, providing control over the partition–replication trade-off. An extension to the basic algorithm, referred to as fault-tolerant Dynamic RePacking, assigns the required minimum number of replicas of each presentation to nodes in a manner that allows load-balancing to be maintained when cluster nodes fail.

The performance of Dynamic RePacking has been shown to approach that of a cloned cluster configuration, in which an entire multimedia archive is replicated on every node, while significantly reducing the storage capacity required on each cluster node. For example, an analysis of the performance of the prototype four-node HammerHead server cluster has shown that, when using Dynamic RePacking, the storage capacity used by the cluster was approximately 25% of that required when the multimedia archive is replicated on every node, with a decrease in achieved service load of just 0.6%. An analysis of the effect of node failures on performance has demonstrated that creating additional replicas of a small number of the most demanding presentations can significantly reduce the effect of node failure on achieved service load.

An event-driven simulation was also used to evaluate a wider range of cluster configurations and client workloads and the accuracy of the simulation was validated by the prototype implementation. Results from this simulation indicate that a 64-node cluster, using Dynamic RePacking to assign each presentation to at least two cluster nodes, would use just 3.5% of the storage capacity required to replicate an archive on every node, with a decrease in achieved service load of approximately 1%.

7.1 Future Work

Although only basic cluster storage configurations have been examined in this thesis, the HammerHead object model, used to represent the state of a multimedia server cluster, has been designed to accommodate more complex configurations or hierarchies

including, for example, clusters of nodes with multiple content mount points or clusters of nodes accessing logical shared storage devices in a storage area network. The Dynamic RePacking policy could be used, without significant modification, to perform dynamic replication of multimedia content within more complex storage hierarchies. Such storage configurations would also require appropriate load-balancing policies, to select a suitable cluster node to service each client request.

The effect of significantly increasing the number of nodes or presentations in a HammerHead cluster on the performance of the group communication system and the state-transfer protocol needs to be investigated further. It should be possible to reduce the state information transferred by a new `HammerSource` instance to the cluster-aware layer by introducing a content discovery protocol, requiring `HammerServer` instances to prompt `HammerSource` plug-ins to inform them of the existence of replicas, as the corresponding presentations are requested by clients. A scheme such as this would have the advantage of only including recently accessed presentations in the aggregated cluster state, reducing the state information maintained by the cluster-aware layer as well as the complexity of the Dynamic RePacking algorithm.

The role of the cluster-aware layer in HammerHead might also be expanded to implement, for example, admission control or stream batching policies on a cluster wide basis. HammerHead could also be used in the construction of a multimedia server cluster in which the nodes use different commodity multimedia streaming technologies (for example, combining the use of Windows Media Services and RealNetworks' Helix Universal Server in a single cluster). The cluster-aware layer might also be extended to allow the cluster to be presented as a single, logical multimedia server in a content distribution network (Section 2.4.6).

Dynamic RePacking, in its current form, does not perform *storage balancing*. A simple extension to the Dynamic RePacking algorithm, which would exploit the highly-skewed distribution of requests for each presentation, could use, for example, a round-robin scheme to assign replicas to nodes for all presentations below some low demand threshold, with Dynamic RePacking used to pack the relatively few presentations with

higher demand. A scheme such as this would have the advantage of approximately balancing the storage utilization across each cluster node and giving more control over any loss of harvest when nodes fail. If the chosen threshold is low, the impact on load-balancing will be small.

There is also scope for further work to be done on improving the allocation of additional replicas to increase the availability of a subset of the presentations in an archive. The performance results presented in this thesis indicate that, although increasing the availability of "important" presentations can significantly reduce the impact of a reduction in harvest on yield, if the ranking of presentations by importance changes frequently, the increased replication rate can have a negative impact on performance. A simple improvement might rank presentations by popularity, rather than demand, based on statistics gathered over longer periods of time or on the relative commercial value of each presentation. An alternative scheme might analyze the placement of replicas after each phase of the fault-tolerant Dynamic RePacking algorithm and increase the minimum replica count of individual presentations to limit the estimated loss in yield resulting from the failure of any node or nodes. A scheme such as this would demonstrate the advantage of separating replication for load-balancing from replication to increase availability.

7.2 Final Remarks

The development of HammerHead has demonstrated that a loosely-coupled cluster of commodity PCs can be used effectively to implement a scalable, highly-available on-demand multimedia streaming service, without the need to perform complete replication of a multimedia archive on every cluster node. The use of group communication, together with a flexible dynamic replication policy that separates replication for load-balancing from replication to increase the availability of individual presentations, allows the performance of HammerHead to degrade gracefully as nodes fail. It is hoped

that HammerHead will provide a suitable framework to allow existing and future developments in the provision of on-demand streaming services over the Internet to be implemented on a cluster-wide basis.

Appendix A

Calculating Mean-Time-To-Service-Loss

Calculations for the reliability of different multimedia server models are shown below. These calculations are based on those used by Patterson et al. [PGK88] and Chen et al. [CLG+94] to express the redundancy of RAID storage systems. Reliability is expressed in terms of the *mean-time-to-data loss* or MTTSL. The *mean-time-to-failure* of a single node is denoted $MTTF$. The *mean-time-to-repair* of a node is denoted $MTTR$. In each case, the number of nodes in the server or cluster is initially N.

MTTSL of a parallel multimedia server with no redundancy

The MTTSL of a parallel multimedia server with N nodes and no redundancy is the MTTF of a single node divided by the number of nodes:

$$MTTSL_{parallel} = \frac{MTTF}{N}$$

MTTSL of a parallel multimedia server with parity

The cluster contains N nodes, one of which contains redundant parity information. It is assumed that the time between failures is exponentially distributed and the failure

rate, λ, is $\frac{1}{MTTF}$. Thus, the probability that a node fails some time *after* t seconds from now, is:

$$\begin{aligned} & \int_t^{\infty} \lambda e^{\lambda x}\, \mathrm{d}x \\ = \quad & e^{-\lambda t} \end{aligned}$$

Replacing $\frac{1}{MTTF}$ for λ and $MTTR$ for t gives the probability that a node will fail after $MTTR$ seconds:

$$e^{-\frac{MTTR}{MTTF}}$$

Like [PGK88], the probability that at least one further node failure will occur in the next $MTTR$ seconds after the first is calculated. This can be expressed as:

$$\begin{aligned} & 1 - P(\text{all remaining nodes fail only after } MTTR) \\ = \quad & 1 - \left(e^{-\frac{MTTR}{MTTF}}\right)^{N-1} \\ = \quad & 1 - e^{-\frac{(MTTR)(N-1)}{MTTF}} \end{aligned}$$

Again the same simplifying assumptions as [PGK88] are made:

$$MTTR \ll \frac{MTTF}{N}$$

and

$$1 - e^{-x} \approx x \qquad \text{for } 0 < x \ll 1$$

Hence the following simplified expression for the probability that a further node failure occurs before the first is repaired:

$$\frac{(MTTR)(N-1)}{MTTF}$$

The mean time until the failure of the first node in a cluster with N nodes is $\frac{MTTF}{N}$. The *mean-time-to-service-loss* ($MTTSL$) of a parallel multimedia server with parity (i.e. one redundant node) can now be expressed as:

$$\begin{aligned} MTTSL_{parity} &= MTTSL_{parallel} \cdot \frac{1}{P(\text{second failure before repair of first})} \\ &= \frac{MTTF}{N} \cdot \frac{MTTF}{(MTTR)(N-1)} \\ &= \frac{MTTF^2}{(N)(N-1)(MTTR)} \end{aligned}$$

MTTSL of a parallel multimedia server with two or more redundant nodes

It is assumed that nodes fail independently and that nodes are repaired consecutively such that the failure of X nodes requires $X \times MTTR$ seconds to repair. If two nodes have failed, with the mean-time-to-failure of the two nodes as given in the previous section, the MTTSL of a parallel server with two redundant nodes can now be determined by calculating the probability that a third node fails before the first two can be repaired. This calculation is similar to that in the previous section, but instead of $N-1$ nodes, we now have $N-2$ and the probability of failure of the third node before the first two can be repaired is:

$$\frac{(MTTR)(N-2)}{MTTF}$$

Hence, the MTTSL for a parallel server with two redundant nodes can be defined recursively:

$$\begin{aligned} MTTSL_{dual_redundancy} &= MTTSL_{parity} \cdot \frac{MTTF}{(MTTR)(N-2)} \\ &= \frac{MTTF}{N} \cdot \frac{MTTF}{(MTTR)(N-1)} \cdot \frac{MTTF}{(MTTR)(N-2)} \end{aligned}$$

Giving the expression for RAID 6 redundancy derived by Chen et al. [CLG+94]:

$$MTTSL_{dual_redundancy} = \frac{MTTF^3}{(N)(N-1)(N-2)(MTTR^2)}$$

In general, the MTTSL of a server with h redundant nodes is based on the probability of $h+1$ nodes failing and can be expressed as [GMCB01]:

$$MTTSL_{Ph} = \frac{MTTF^{h+1}}{\left(\prod_{i=0}^{h}(N-i)\right)(MTTR^h)}$$

MTTSL of a cloned multimedia server

In this case, multimedia content is mirrored on every node so $N-1$ failures can occur before data is lost. The MTTDL of such a sever can be expressed as follows:

$$MTTDL_{cloned} = \frac{MTTF^N}{N!(MTTR^{N-1})}$$

Bibliography

[ACK+97] T. Anker, G. V. Chockler, I. Keidar, M. Rozman, and J. Wexler. Exploiting group communication for highly available video-on-demand services. In *Proceedings of the 13th International Conference on Advanced Science and Technology (ICAST97) and the 2nd International Conference on Multimedia Information Systems (ICMIS 97)*, Schaumburg, Illinois, USA, April 1997.

[ACM96] ACM. Comunications of the ACM. *Special issue on Group Communications Systems*, 39(4), April 1996.

[ADK99] T. Anker, D. Dolev, and I. Keidar. Fault tolerant video on demand services. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, Austin, Texas, USA, June 1999.

[AKEV01] J. M. Almeida, J. Krueger, D. L. Eager, and M. K. Vernon. Analysis of educational media server workloads. In *Proceedings of the 11th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Port Jefferson, New York, USA, June 2001.

[AS98] S. Acharya and B. Smith. Characterizing user access to videos on the world wide web. In *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN)*, San Jose, California, USA, January 1998.

[AS00] S. Acharya and B. Smith. MiddleMan: A video caching proxy server. In *Proceedings of the 10th International Workshop on Network and Operating*

System Support for Digital Audio and Video (NOSSDAV), Chapel Hill, North Carolina, USA, June 2000.

[BB96] C. Bernhardt and E. Biersack. The server array: A scalable video server architecture. In *High-Speed Networking for Multimedia Applications*, pages 103–126. Kluwer, March 1996.

[BBD+96] W. Bolosky, J. Barrera, R. Draves, R. Fitzgerald, G. Gibson, M. Jones, S. Levi, N. Myhrvold, and R. Rashid. The tiger video fileserver. In *Proceedings of 6th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Zushi, Japan, April 1996.

[BBS99] P. Berenbrink, A. Brinkmann, and C. Scheideler. Design of the presto multimedia storage network. In *Proceedings of the International Workshop on Communication and Data Management in Large Networks*, Paderborn, Germany, October 1999.

[BDH03] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, March/April 2003.

[BFD97] W. J. Bolosky, R. P. Fitzgerald, and J. R. Douceur. Distributed schedule management in the tiger video fileserver. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 212–223, Saint-Malo, France, 1997.

[Bir96] K. P. Birman. *Building Secure and Reliable Network Applications.* Manning Publications Company and Prentice Hall, 1996.

[Bre01] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July/August 2001.

[Bri95] T. Briscoe. DNS support for load balancing. IETF RFC 1794, April 1995.

[BSS00] A. Brinkmann, K. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the 12th*

ACM Symposium on Parallel Algorithms and Architectures, pages 119–128, Bar Harbor, Maine, USA, July 2000.

[Buf94] J. F. Koegel Buford. *Multimedia Systems.* ACM Press / Addison-Wesley, 1994.

[CDK01] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, 3$^{\text{rd}}$ edition, 2001.

[CGK$^+$01] C. D. Cranor, M. Green, C. Kalmanek, D. Shur, S. Sibal, J. Van der Merwe, and C. J. Sreenan. Enhanced streaming services in a content distribution network. *IEEE Internet Computing*, 5(4):66–75, July/August 2001.

[CGL00] C. Chou, L. Golubchik, and J. Lui. Striping doesn't scale: How to achieve scalability for continuous media servers with replication. In *Proceedings of 20th International Conference on Distributed Computing Systems*, pages 64–71, Taipei, Taiwan, April 2000.

[Che94] A. L. Chervenak. *Tertiary Storage: An Evaluation of New Applications.* PhD thesis, University of California at Berkeley, 1994.

[CKV01] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, December 2001.

[CLG$^+$94] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–188, June 1994.

[CWVL01] M. Chesire, A. Wolman, G. M. Voelker, and H. M. Levy. Measurement and analysis of a streaming-media workload. In *Proceedings of 3th USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, USA, March 2001.

[DM96] D. Dolev and D. Malkhi. The transis approach to high availability cluster communication. *Communications of the ACM*, 39(4), April 1996.

[DS95a] A. Dan and D. Sitaram. Dynamic policy of segment replication for load-balancing in video-on-demand servers. *ACM Multimedia Systems*, 3(3):93–103, July 1995.

[DS95b] A. Dan and D. Sitaram. An online video placement policy based on bandwith to space ratio (BSR). In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 376–385, San Jose, California, USA, May 1995.

[DSS94] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Proceedings of the 2nd ACM International Multimedia Conference*, pages 15–23, San Francisco, California, USA, October 1994.

[Fer03] A. Ferreira. Optimizing Microsoft Windows Media Services 9 Series. Technical article, Microsoft Corporation, February 2003.

[FGC$^+$97] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, San Malo, France, October 1997.

[FGM$^+$99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. IETF RFC 2616, June 1999.

[FK01] A. Fekete and I. Keidar. A framework for highly available services based on group communication. In *Proceedings of the International Workshop on Applied Reliable Group Communication (WARGC)*, pages 57–62, Phoenix, Arizona, USA, April 2001.

[GB03] T. Gill and B. Birney. *Microsoft Windows Media Resource Kit.* Microsoft Press, 2003.

[GBW97] C. Griwodz, M. Bär, and L. C. Wolf. Long-term movie popularity models in video-on-demand systems: or the life of an on-demand movie. In *Proceedings of the 5th ACM International Multimedia Conference*, pages 349–357, Seattle, Washington, USA, November 1997.

[GMCB01] L. Golubchik, R. R. Muntz, C.-F. Chou, and S. Berson. Design of fault-tolerant large-scale VOD servers: With emphasis on high-performance and low-cost. *IEEE Transactions on Parallel and Distributed Systems*, 12(4), April 2001.

[GS00] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proceedings of the 16th International Conference on Data Engineering*, pages 3–12, San Diego, California, USA, February/March 2000.

[Guh99] A. Guha. The evolution to network storage architectures for multimedia applications. In *Proceedings of the IEEE Conference on Multimedia Computing Systems*, pages 68–73, Florence, Italy, June 1999.

[Hay97] M. Hayden. *The Ensemble System.* PhD thesis, Department of Computer Science, Cornell University, 1997.

[HCS98] K. A. Hua, Y. Cai, and S. Sheu. Patching: A multicast technique for true video-on-demand services. In *Proceedings of the 6th ACM International Multimedia Conference*, pages 191–200, Bristol, UK, September 1998.

[HJ98] M. Handley and V. Jacobson. Session description protocol (SDP). IETF RFC 2327 (proposed standard), April 1998. Available at http://www.ietf.org/rfc/rfc2327.txt.

[HP03] J. L. Hennessy and D. L. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 3rd edition, 2003.

[HS96] R. L. Haskin and F. B. Schmuck. The Tiger Shark file system. In *Proceedings of the 41st IEEE Computer Society International Conference (COMPCON '96)*, pages 226–231, Santa Clara, California, USA, March 1996.

[HvR97] M. Hayden and R. van Renesse. Optimising layered communication protocols. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, page 169, Portland, Oregon, USA, 1997.

[Knu98] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

[Lee98] J. Y. B. Lee. Parallel video servers: A tutorial. *IEEE Multimedia*, 5(2):20–28, April–June 1998.

[Lit61] J. D. C. Little. A proof of the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.

[LL97] W. Liao and V. O. K. Li. The split and merge protocol for interactive video-on-demand. *IEEE Multimedia*, 4(4):51–62, October – December 1997.

[LLG98] P. W. K. Lie, J. C. S. Lui, and L. Golubchik. Threshold-based dynamic replication in large-scale video-on-demand systems. In *Proceedings of 8th International Workshop on Research Issues in Data Engineering (RIDE)*, Orlando, Florida, USA, February 1998.

[LV94] T. D. C. Little and D. Venkatesh. Prospects for interactive video-on-demand. *IEEE Multimedia*, 1(3), Autumn/Fall 1994.

[LV95] T. D. C. Little and D. Venkatesh. Popularity-based assignment of movies to storage devices in a video-on-demand system. *ACM Multimedia Systems*, 2(6):280–287, January 1995.

[Mic00] Microsoft Corporation. Network load balancing technical overview. White paper, Microsoft Corporation, January 2000.

[Mic] Microsoft Corporation. Windows media 9 series SDK. Information available at http://www.microsoft.com/windows/windowsmedia/9series/sdk.aspx.

[MS02] H. Ma and K. G. Shin. Multicast video-on-demand services. *ACM SIGCOMM Computer Communication Review*, 32:31–43, January 2002.

[PGK88] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, USA, 1988.

[Rea03] RealNetworks. *Helix Universal Server Administration Guide*, May 2003.

[Red95] A. L. N. Reddy. Scheduling and data distribution in a multiprocessor video server. In *Proceedings of the 2th IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, Washington, D.C., USA, May 1995.

[RS92] L. A. Rowe and B. C. Smith. A continuous media player. In *Proceedings of 3th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, La Jolla, California, USA, November 1992.

[SDY93] D. Sitaram, A. Dan, and P. S. Yu. Issues in the design of multi-server file systems to cope with load skew. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 214–223, San Diego, California, USA, January 1993.

[SGB98] D. N. Serpanos, L. Georgiadis, and T. Bouloutas. MMPacking: A load and storage balancing algorithm for distributed multimedia servers. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(1):13–17, February 1998.

[SM98] J. R. Santos and R. Muntz. Performance analysis of the RIO multimedia storage system with heterogeneous disk configurations. In *Proceedings of the*

6th ACM international conference on Multimedia, pages 303–308, Bristol, UK, September 1998.

[SRL98a] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (RTSP). IETF RFC 2326 (proposed standard), April 1998.

[SRL98b] H. Schulzrinne, A. Rao, and R. Lanphier. RTP: A transport protocol for real-time applications. IETF RFC 2326, April 1998.

[SRT99] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'99)*, pages 1310–1319, New York, New York, USA, March 1999.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition edition, 1996.

[VAM+02] E. Veloso, V. Almeida, W. Meira, A. Bestavros, and S. Jin. A hierarchical characterization of a live streaming media workload. In *Proceedings of the Second ACM SIGCOMM Workshop on Internet Measurment*, pages 117–130, Pittsburgh, Pennsylvania, USA, November 2002.

[Vay98] A. Vaysburd. *Building Reliable Interoperable Distributed Objects with the Maestro Tools*. PhD thesis, Cornell University, 1998.

[vBM96] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication toolkit. *Comminications of the ACM*, 39(4):76–83, April 1996.

[Ver02] D. C. Verma. *Content Distribution Networks: An Engineering Approach*. Wiley-Interscience, 2002.

[VR97] N. Venkatasubramanian and S. Ramanathan. Load management in distributed video servers. In *Proceedings of the International Conference on Distributed Computing Systems*, Baltimore, Maryland, USA, May 1997.

[Vv94] W. Vogels and R. van Renesse. Support for complex multimedia applications using the horus system. On-line html document, Department of Computer Science, Cornell University, Ithaca, New York, USA, 1994.

[WL97] P. C. Wong and Y. B. Lee. Redundant arrays of inexpensive servers (RAIS) for on-demand multimedia services. In *Proceedings ICC 97*, pages 787–792, Montréal, Québec, Canada, June 1997.

[WV01] X. Wei and N. Venkatasubramanian. Predictive fault-tolerant placement in distributed video servers. In *IEEE International Conference on Multimedia and Expo 2001 (ICME 2001)*, Tokyo, Japan, August 2001.

[WYS95] J. L. Wolf, P. S. Yu, and H. Shachnai. DASD dancing: A disk load balancing optimization scheme for video-on-demand computer systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS '95)*, pages 157–166, Ottawa, Ontario, Canada, May 1995.

[Zip49] G. K. Zipf. *Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology.* Addison-Wesley, 1949.

[ZX02a] X. Zhou and C.-Z. Xu. Optimal video replication and placement on a cluster of video-on-demand servers. In *Proceedings of the 31st International Conference on Parallel Processing (ICPP'02)*, pages 547–555, Vancouver, British Colombia, Canada, August 2002.

[ZX02b] X. Zhou and C.-Z. Xu. Request redirection and data layout for network traffic balancing in cluster-based video-on-demand servers. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, pages 127–134, Fort Lauderdale, Florida, USA, April 2002.