A Redundant Hierarchical Structure for a Distributed Continuous Media Server*

Cyrus Shahabi, Mohammad H. Alshayeqi, and Shimeng Wang  
Integrated Media Systems Center and  
Computer Science Department  
University of Southern California  
Los Angeles, California 90089  
[cshahabi, alshayeq, shimeng]@cs.usc.edu

March 21, 1997

Abstract

The growing number of digital audio and video repositories has resulted in a desperate need for effective techniques to deliver data to users in a timely manner. Due to geographical distribution of users, it is not cost effective to have a centralized media server. In this paper, we investigate issues involved in the design of a distributed video server (DVS) to support movie-on-demand (MOD) application. We propose a redundant hierarchical (RedHi) architecture for DVS where the nodes are continuous media servers and the edges are dedicated network lines. With RedHi, each node has two or more parents. We show that the redundant links in RedHi yield a more reliable and efficient system. Our simulation results demonstrate that RedHi can tolerate a single link failure with no degradation in performance while with pure hierarchy almost 2.5% of requests are rejected due to the failure. In normal mode of operation, RedHi outperforms pure hierarchy significantly (160% improvement on the average when counting the number of rejections). In the context of RedHi, we also propose and evaluate alternative object management policies, resource reservation strategies, and load balancing heuristics. Furthermore, we investigate two adaptive buffering mechanisms that employ client side cache to improve system utilization.

*This research has been funded by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center (NSF grant EEC-9529152) with additional support from the Annenberg Center for Communication at the University of Southern California and the California Trade and Commerce Agency.
1 Introduction

The growing number of digital audio and video repositories has resulted in a desperate need for effective techniques to deliver data to users in a timely manner. Due to geographical distribution of users, it is not cost effective to have a centralized media server. Distributed media servers can be employed by numerous application domains, e.g., digital libraries, health-care information systems, and educational applications to name a few.

In this paper, we investigate issues involved in the design of a distributed video server (DVS) to support movie-on-demand (MOD) application. With MOD, the request of a user to watch a (say 2 hour) movie should be served within a reasonable amount of delay (say 2 minutes). DVS consists of a number of continuous media servers, such as *Mitu* [GZS+97] or *Fellini* [MNO+97], connected to each other via dedicated network lines. Unlike the customers of a video rental store, the users of DVS do not have to drive to the store to rent a movie nor need to worry about the availability of their desired movies. As compared to *pay-per-view* TV, the users of DVS are provided with a larger selections and can watch their selected movies at any desired time. In addition, advertisers can benefit from users' profiles to better direct their commercials to potential customers. This type of narrow-cast advertising is very attractive to advertisers and hence it can be employed to amortize the system cost among them. The users can either watch a movie free of charge by also watching a number of customized commercials or pay for the movie and watch it commercial free.

In summary, DVS is a more flexible and potentially cheaper service as compared to *pay-per-view* and video rental stores. By collectively supporting the customers of both of these services, the success of DVS is trivial.

There are two alternative designs to support MOD as opposed to the DVS design: 1) a single large centralized server, and 2) a number of independent servers. The geographical disbursement of users results in a high communication cost for the first approach. The bandwidth requirement for this approach is estimated to be as high as 1.54 Pb/s (Peta-bit per second) for the continental United States [NPSS95]. With the second approach, each server must be large enough to accommodate all
the movies. This requires local servers with large storage space and imposes an extra overhead of maintaining each server (i.e. insertion of new movies and deletion of older ones). Although DVS still requires large national servers that can store a rich set of movies, objects can be cached on smaller local servers closer to the users in order to reduce the communication cost. Moreover, the number of large servers in this design is much less than that of the independent servers design.

To strike a compromise between the communication cost and the server cost, some studies [BP96, NPSS95] have proposed a hierarchical network topology for a DVS. In this paper, however, we propose a redundant hierarchy (RedHi) architecture. With RedHi, each node has two or more parents. This not only makes the system more fault-tolerant (if a connection fails, the client will not become isolated), but also improves load balancing (each node has a choice to where to obtain the data in order to balance the load). We believe that the improvement in the performance and reliability of the system justifies the extra cost of the redundant links.

Assuming the RedHi architecture, we address the following issues in this paper:

- Distributed information management: Each node of RedHi (i.e., a server) stores static information about all the objects such as object size and display format. Nodes do not, however, keep track of dynamic information such as the locations of the objects. We show that an object can be located and retrieved in a distributed manner. Distributed information management eliminates the possibility of bottlenecks and makes the system more fault-tolerant.

- Load balancing: Since nodes of RedHi have more than one parent, each node might be able to obtain an object through multiple paths. We propose a number of heuristics to find the best path to balance the load. We show our heuristics can be implemented in RedHi with low complexity.

- Resource reservation: When multiple nodes are simultaneously exercising their options to find best possible paths, different resource reservation strategies must be employed to prevent collisions. We propose three reservation strategies: optimistic, pessimistic and server-lock.

- Adaptive buffering mechanisms: We propose two adaptive buffering mechanisms that employ
client side memory to improve the system utilization. The *Eager Pipelining* mechanism can utilize uncommitted resources to speed up object delivery and hence enabling the system to free reserved resources earlier. The *Lazy Pipelining* can be employed to serve a request which requires more bandwidth than what is available.

- RedHi configuration: Determining the number and the capacity of servers and links in RedHi as well as its connection topology is a complex task. While we do not address this issue thoroughly, we provide some guidelines to help a designer for configuring RedHi.

- Performance evaluation via a simulation study: We compared RedHi with pure hierarchical architecture in both normal mode of operation and in the presence of failure. Our simulation studies demonstrated an average of 160% improvement with RedHi in normal mode of operation. When the system is not over-utilized (i.e., load ≤ 100%), a single link failure resulted an average of 2.5% rejections with pure hierarchy while RedHi rejected no request. In addition, we evaluated the performance of our replacement policies and path finder heuristics.

The rest of this paper is organized as follows. Sec. 2 covers some related work on distributed server architectures. The RedHi architecture is formally defined in Sec. 3. In Sec. 4, our object caching and replacement policies are explained. The heuristics for balancing the load in RedHi are discussed in Sec. 5, and Sec. 6 presents the adaptive buffering mechanisms. In Sec. 7, our simulation experiments are described. Finally, Sec. 8 concludes the paper and provides an overview on our future plans.

## 2 Related Work

A significant amount of research was conducted on centralized multimedia servers (e.g., [GZS+97, GVK+95, RVR92, MNO+97, CL93, HLLD97, LPS94]), as well as high-speed networks (e.g., [Pry95, RRM+93, IA93]). The challenge, however, is to marry these two technologies to achieve cost effective delivery of multimedia objects to users.
A number of distributed server architectures were proposed in [BP96, NPSS95, LLQW96, DVV94]. Frequently the proposed network architecture in these studies is assumed to be a $d$-ary tree with each node representing a switch (see Fig. 1(a)). Moreover, at each node (switch) there exists a multimedia server which is employed to cache a subset of the objects. At the root of the tree sits what is referred to as a regional or national server. A national server should be large enough to store all the available objects. A regional server, however, is connected via another network to other regional servers which collectively store all the objects available in the system. At the lowest level of the hierarchy are the head-end nodes to which users are connected. Asymmetric digital subscriber line (ADSL) seems to be the strongest candidate for connecting users to these head-end nodes [LV94]. The higher levels of the hierarchy, however, are typically based on the asynchronous transfer mode (ATM) technology.

In this paper, we are extending the above architecture into a redundant hierarchical architecture to achieve a better fault-tolerance and load balancing. The work in [BP96, NPSS95, LLQW96, DVV94] concentrated on the communication aspects of this distributed server architecture while our major focus is to develop methods and policies for the distributed management of objects in this architecture. It is also important to point out that although DVS has some similarities to traditional distributed information systems (e.g. [Sim96]), it is mainly designed for uninterrupted delivery of continuous media objects, something that has not been considered in traditional distributed information systems.

3 System Architecture

The main motivation behind proposing hierarchical DVS (pure or redundant), as mentioned earlier, is to reduce the communication cost. Communication cost can be defined as the cumulative bandwidth requirement of the system. Allowing users to connect to local head-ends, as opposed to having them connected to a centralized server, can significantly reduce the cumulative bandwidth requirement of the system.
In hierarchical DVS, uninterrupted delivery of continuous media object is not a trivial task. In this paper, we assume a pipelined object delivery. That is, the first block of data reaches the user after an initial delay; all consequent blocks will then arrive in a timely fashion with no interruptions. To illustrate, suppose that a user connected to node F (see Fig. 1b) requested an object \( O_i \) (a 2 hour MPEG-2 movie with a bandwidth requirement of 2 Mb/s). Node F needs to locate \( O_i \) and select a path to retrieve it (locating an object and selecting a path will be discussed in Sec. 5). Suppose that the highlighted path in Fig. 1b was selected to retrieve \( O_i \) from node A. Prior to the initiation of the display, the system needs to reserve the required bandwidth (2 Mb/s) at node A and all the links participating in the path. Note that it is not essential (but possible) to materialize \( O_i \) at any of the nodes through the path (including node F). After reserving the resources along the path, a pipeline is established. That is, the display of \( O_i \) starts after the arrival of its first block to node F and continues with no interruptions for 2 hours. In Sec. 6, we show how we relax the resource reservation to reserve less or more bandwidth than what is required for object display.

In addition to eliminating the one parent restriction in RedHi, as opposed to pure hierarchy in [BP96, NPSS95], we do not assume that nodes at the same level have the same amount of resources (server bandwidth and storage capacity). Nor we assume that links connecting nodes of level \( k \) to level \( k-1 \) of the hierarchy have an identical bandwidth. Therefore, when the number

---

Figure 1: Illustration of Pure Hierarchy & RedHi
of users of a head-end increases, the system can be expanded by adding more resources at the head-end (e.g., more disk space), adding a new link, increasing the bandwidth of existing links or applying a combination of the three.

3.1 Formal definition of RedHi

The concept of Leveled Graph will be defined as a generalization of bipartite graph. RedHi will then be defined as a special case of leveled graph.

**Definition 3.1:** Leveled Graph \( \mathcal{L} \mathcal{G} = (\mathcal{V}, \mathcal{E}) \) is a graph where \( \mathcal{V} \) is the set of vertices and \( \mathcal{E} \) is the set of edges, and it satisfies the following two conditions:

1. There exists a partition \( \{\mathcal{V}_i \mid i \in 1..n, n \geq 1\} \) of \( \mathcal{V} \), such that if \( n > 1 \) then \( \forall e \in \mathcal{E}, \) where \( e = (u, w), \exists j \in 2..n, \) such that \( u \in \mathcal{V}_j \) and \( w \in \mathcal{V}_{j-1} \). That is, edges connect vertices of neighboring levels. Here, \( w \) is called the parent node of \( u \), and \( u \) is called the child node of \( w \).

2. If \( n > 1 \) then \( \forall v \in \mathcal{V}_i, \) where \( i \in 2..n, \exists e = (u, w) \in \mathcal{E} \) such that \( v = u \). \( \forall v \in \mathcal{V}_1, \exists e = (u, w) \in \mathcal{E} \) such that \( v = w \). That is, every vertex in \( (\mathcal{V} - \mathcal{V}_1) \) has a parent and every vertex in \( \mathcal{V}_1 \) has a child. 

A leveled graph can be expressed as \( \mathcal{L} \mathcal{G} = (\mathcal{V}, \mathcal{E}, n) \), where \( n \) is the number of levels in the graph. Level \( \mathcal{V}_1 \) contains vertices that are called root or top level nodes. Nodes that have no children (i.e., leaf nodes) are termed head-ends.

*Pure Hierarchy* (or Tree) is an \( \mathcal{L} \mathcal{G} \) that has one and only one root and every node except the root has one and only one parent. *Redundant Hierarchy* is an \( \mathcal{L} \mathcal{G} \) that has at least two roots and every node except the roots have at least two parents.

**Definition 3.2:** Distributed Server Architecture \( \mathcal{D} \mathcal{S} \mathcal{A} \) is an \( \mathcal{L} \mathcal{G} = (\mathcal{V}, \mathcal{E}, n) \) where

1. A link of \( \mathcal{D} \mathcal{S} \mathcal{A} \) is an edge of its corresponding \( \mathcal{L} \mathcal{G} \) with two parameters: \( \text{MaxB} \) and \( B \), which are the maximum and available bandwidth of the link, respectively.
2). A server of $D\Sigma A$ is a vertex of its corresponding $LG$ with five parameters: $MaxB$ (maximum bandwidth), $B$ (available bandwidth), $MaxC$ (maximum capacity), $C$ (available capacity), and $ObjInfo$ (objects information); where each object has six parameters: $Name$ (object name), $Time$ (recent access time for the object), $Freq$ (accumulated access frequencies), $Flag$ (if the object exists in the server), $Cont$ (content of the stored object), and $Size$ (object size).

Notation: $D\Sigma A$ can either be a Pure Hierarchy or a Redundant Hierarchy ($RedHi$).

A path in $D\Sigma A = (V, E, n)$ is a sequence $\langle v_0, v_1, ..., v_k \rangle$ of servers, where $k$ is the length of the path, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, ..., k$. Furthermore, for a path to be able to deliver the requested object in $D\Sigma A$, termed delivery path, the following condition must be satisfied: $\forall j \in 0..(k - 1), v_j$ be the child of $v_{j+1}$.

### 3.2 Distributed information management

Each node of a DSA stores a table that contains static information about all the objects such as object size and display format. In addition, each node keeps track of the local access frequency of each object. That is, the number of instances that the children of this node have requested each object. These frequencies will be used by the replacement policies that will be discussed in Sec. 4. Nodes do not, however, keep track of dynamic information such as the locations of the objects. When a node receives a request for an object, it either responds to the requester indicating that it has the object and able to deliver it, or forward this request to its parents. The request will continue propagating up in the hierarchy until the object is found. One alternative is to keep the information about all objects at a centralized location. In this case, every request and update should visit the centralized server and hence it might become a bottleneck. Another alternative is that each node maintains information about the location of every object in the system. This results in a higher maintenance overhead and may suffer from inconsistency problems.
3.3 Some Issues on Configuring RedHi

Configuring RedHi is not a trivial task. Based on the system objectives, a system designer should decide on the size of the nodes (servers capacity $MaxC$ and bandwidth $MaxB$), distribution of these nodes, the number of links between the nodes, the bandwidth $MaxB$ of the links, as well as the number of levels of the hierarchy. In this section, we attempt to highlight some of the issues that the designer should consider when configuring RedHi. We propose three approaches to configure RedHi. The first approach configures RedHi based on the number of users. The second approach uses a geographical convention to configure RedHi, and the third is a hybrid of the first two approaches.

With the first approach, to decide the servers size, the designer may use guidelines similar to the ones used by the current cable TV (CATV) tree broadcast infrastructure. Since the number of rejected requests is proportional to the number of users served by the head-end, we can limit the number of users of a head-end (e.g., less than 1000 users in CATV [NS94]) while keeping the size of all head-ends constant. Although a good load indicator, the number of users should not be the only factor that a designer considers when deciding on the size of the head-ends. This is because, a company requesting three training videos a day should not be treated as a residential user that requests two movies a week. In addition, an ethnically diversified population is likely to have a larger selection preference than a population of the same ethnic background. Other social and economical aspects impact the user demand. Therefore, the user profile can play a role in the design process. Another drawback of this approach is the fact that using a fixed size head-end may result in a user population that is scattered over a large geographical area increasing the communication cost.

A similar argument can be made when deciding on links bandwidth. The designer should take into consideration the number of users as well as users profile. In addition, she should consider the total number of objects in the system. The more objects the system stores the more diversified the requests will be, and thus the harder it is to cache objects closer to users.
With the second approach a geographical convention is used when deciding on the number of levels of the system. Since the physical distance has a significant impact on the overall communication cost of the system, it would be wise to consider it during the system design. RedHi can be divided into levels with each node in a level designated to serve a specific geographical area. A head-end at the lowest level of the hierarchy can be designated to serve a limited geographical area. A node at the second level serves an entire city (a group of areas). The third level of the hierarchy is designated as the county level and so on. The problem with this approach is that cities, counties, and states vary in their sizes and population density. New York, for example, spans a much smaller area than Los Angeles, but is likely to have more customers than Washington D.C. Defining a convention like area, city, and county considers the physical distance – to some extent – but the final design may suffer due to treating areas, cities, and counties that have different characteristics in the same manner.

Finally, a hybrid approach employs both physical distance and users profile. Users profile along with the population of a certain area can be used to estimate the area's demand. The estimated demand can then be used to decide on the size of the head-end serving this area. Servers on the second level of the hierarchy are positioned according to their physical proximity to their supported head-ends. As we go up on the hierarchy, we try to minimize communication cost by grouping nodes that are physically close to each other in a "sub-hierarchy" and allow only the nodes in the highest level of this sub-hierarchy to be connected to the rest of the system. This approach considers server cost by selecting head-end size according to the user’s population and profile. Moreover, it considers communication cost by grouping nodes, according to their physical proximity, into sub-hierarchies and minimizing the number of links from these sub-hierarchies to the rest of the system. In this paper, we do not investigate this issue any further and we assume that the configuration of DSA is given.
4 Objects Management

The system strives to minimize communication cost, by caching objects closer to users. As mentioned earlier, head-end servers can accommodate only a small subset of the rich selection of objects in the system. Therefore, a replacement policy is needed when the storage space of head-ends is exhausted. The problem is very similar to the memory hierarchy problem. The popularity of the object is usually the main factor to consider when it is cached. A semi-dynamic approach for object replacement was proposed in [BP96]. They assumed that it is possible to anticipate the popularity of some objects before their insertion to the system. This anticipated popularity is then used to decide the residence of the object. The actual popularity is then checked periodically (e.g., daily) and whenever the popularity drops, the object is replaced by a more popular one.

We, however, propose a dynamic approach that is similar to the least frequently used (LFU) replacement policy used in memory hierarchy. In this policy, objects are not assumed to have a predicted popularity, and are always inserted into the root nodes. Instead of examining the popularity periodically, we check it every time an absent object is requested by the children of a node. If 1) the popularity of the absent object exceeds the popularity of any of the objects available at that node, 2) the removal of the less popular object(s) frees enough space to accommodate the new object, and 3) the server has enough bandwidth \( B \) to store the new object, then a replacement takes place. Otherwise, the object is delivered to the requester without caching (i.e., a copy of the object is not materialized at the server).

To illustrate, assume that object \( O_i \) is requested by a user connected to node F (Fig. 1). Suppose we somehow located \( O_i \) in node A and selected the highlighted path to display the object to the user (see Sec. 5). Node F, as well as every other nodes in the path from A to F, should compare the popularity of object \( O_i \) with the popularity of all the objects resident at that node. A replacement occurs if three conditions are met; 1) the popularity of \( O_i \) exceeds the popularity of one or more object(s) available at that node \( (Freq(\text{the old objects}) < Freq(\text{the new object}) \text{ if LFU}) \)  2) the removal of the less popular object(s) frees enough space to accommodate the new object, and 3)
the server has enough bandwidth ($B$) to store the new object. If any of these conditions is violated, the object is forwarded without caching. The use of a dynamic replacement policy makes the system more adaptive to sudden changes in objects popularity which is typically the case in movie industry.

It is important to point out that when a node decides to cache $Oi$, it will follow the $PDF$ dataflow paradigm introduced in [GDS95]. That is the caching and the delivery of $Oi$ to the lower node are performed in parallel.

New objects can be inserted into the system by storing them on the root nodes. As the popularity of these objects increases, the system replacement policy will cache them closer to the users. In Sec. 7, the performance of LFU is compared with that of LRU. The main difference between these two replacement policies is the way object popularity is calculated. LFU uses the frequency of access as its popularity indicator where LRU uses the latest time that the object was referenced as its popularity indicator.

5 Load Balancing

The existence of redundant links in RedHi introduces a new problem that was not there with the pure hierarchy (PHi) architecture. With PHi, when a node receives a request for an object not residing at that node, it propagates the request to its parent. Since each node has only one parent, this request continues to traverse up the tree until it reaches a node that contains the object, possibly the root. With RedHi, each node has two or more parents and hence it may have more than one source to retrieve the object. There might even be alternative paths to each of these sources. Therefore, a policy is required for choosing the best source-path combination.

One can always use a priority scheme that forces each node to select one of its parent as the primary parent. In this case, if the primary parent is saturated or close to saturation, other parents can be utilized. Although simple, this scheme does not fully explore the system’s load balancing potential. This is because it may lead to the overloading of the primary parent while leaving other
parents under-utilized. We propose a number of heuristics to choose the least loaded path. These heuristics consider the load of all the participating links and servers.

5.1 Cost Functions

To balance the load one can choose the least loaded path among all the candidate paths. There are many ways to quantify the load of a path. One method is to take the summation of the loads (i.e., reserved bandwidth) of all the participating nodes and links in the path. In this case longer paths, with more links, are more likely to have higher loads. Trivially with this approach longer paths are penalized.

An alternative method would be to take the average of the loads of all participating nodes and links in the path. This method does not provide shorter paths with any advantage. However, if a path has a number of lightly loaded links and nodes but a single nearly saturated node, this method would choose this path over an averagely loaded path. This may saturate the heavily loaded node and prevent it from serving other requests while the nodes in the other path are less loaded. Therefore, we have decided to choose the cost of the most loaded component in the path as the cost of the path. That is, the node or the link which is most loaded determines the load of its corresponding path.

We propose three cost functions corresponding to our three heuristics to measure the load of a node or a link in order to select the least loaded path. The first cost function FreeBW uses the available bandwidth \( B \) of servers or links as the load indicator\(^1\). The disadvantage of FreeBW is that it tends to select nodes with higher bandwidth in the higher levels of the hierarchy over nodes at the lower levels with less bandwidth. Therefore, a large node operating at fifty percent of its maximum bandwidth is going to be selected over a smaller node, lower in the hierarchy, with the same load percentage. This can unfairly saturate higher nodes and might yield a higher communication cost since most of the objects will be retrieved from higher levels of the hierarchy.

\(^1\)In this case, the cost function is \(-B\).
To overcome this drawback, a second cost function $\text{RatioBW}$ is proposed. Instead of using available bandwidth as load indicator, $\text{RatioBW}$ uses the available bandwidth ratio ($\text{RatioBW} = \frac{B}{\text{MaxB}}$) to measure the load.

The third cost function $\text{UserBW}$ employs an alternative method to eliminate the disadvantage of $\text{FreeBW}$. $\text{UserBW}$ divides the available bandwidth of a node by the number of users served collectively by the node and its children ($\text{UserBW} = B/\text{total number of users served by the node}$). Similarly the $\text{UserBW}$ of a link $e = (u, w)$ where $u \in V_j$ and $w \in V_{j-1}$ is defined as the available bandwidth of $e$ divided by the number of user served collectively by $u$. These three cost functions are compared in Sec. 7, and the result show the superiority of $\text{RatioBW}$.

### 5.2 Object Locating Methods

As mentioned earlier, the goal of object locating methods is to find a node that contains the requested object and is capable of delivering it. We investigate two methods to locate objects in RedHi: $\text{Local}$ and $\text{Global}$. With $\text{Local}$, when a node $Si$ receives a request for an object $Oi$, it first examines if $Oi$ is available locally and if it has the required server bandwidth to deliver $Oi$; if both are true, then the node responds to the requester declaring itself as a possible source for $Oi$. Otherwise, the request is forwarded to $Si$’s parents.

Instead, with $\text{Global}$, a node $Si$ forwards the request to its parents even if $Oi$ is available locally. By doing so, $\text{Global}$ hopes to find a node, higher in the hierarchy, which can deliver $Oi$ at a lower cost.

By stopping the search after locating the first copy of the object in a path, $\text{Local}$ attempts to utilize objects that are closer to user. Retrieving closer objects can result in significant reduction in communication cost. On the other hand, if we keep on selecting objects in the lower levels of the hierarchy, we run the risk of overloading these nodes while leaving nodes in the higher level under-utilized. $\text{Global}$ attempts to balance the load between nodes at the higher and lower levels of the hierarchy.
We have developed efficient heuristics to calculate the costs of the paths, compare these costs and select the least expensive path. The main structure of the heuristics is identical and can be divided into two phases: upward (Fig. 2) and downward (Fig. 3) phases. The difference between these heuristics is due to their use of different cost functions (i.e., FreeBW, UserBW, RatioBW).

\[\text{upward}(\text{node}, <\text{current}\_\text{cost}, \text{object}, \text{path}>)\]

let \(\text{path} = \{v_0, \ldots, v_k\}\), then \(\exists i\), such that \(\text{node} = \text{path}.v_i\)

let \(b\text{Value} = (\text{node}.B > \text{Movie Bandwidth} \text{ and } \text{node}.\text{object}.\text{Flag})\)

if \((b\text{Value})\)

set \(\text{current}\_\text{cost} = \text{max}(\text{current}\_\text{cost}, \text{cost-function}(\text{node}))\)

if (using ‘Local’ method)

send message \(<\text{current}\_\text{cost}, \text{object}, \text{path}>\) to \(\text{node} \text{path}.v_{i-1}\) (\(i > 0\)),
or \(\text{node} (i == 0)\)

if (using ‘Global’ method)

store \(\text{current}\_\text{cost}\)

if (not \(b\text{Value}\)) or if (using ‘Global’ method)

for (all the links that \(\text{linke.u} == \text{node.v}\)) (let \(e = (u, v)\))

if (\(\text{link}.B > \text{Movie Bandwidth}\))

set \(\text{current}\_\text{cost} = \text{max}(\text{current}\_\text{cost}, \text{cost-function}(\text{link}))\)

set \(\text{path}.v_{i+1} = \text{linke.w}\)

send message \(<\text{current}\_\text{cost}, \text{object}, \text{path}>\) to \(\text{linke.w}\)

Figure 2: The upward process

\[\text{downward}(\text{node}, <\text{current}\_\text{cost}, \text{path}>)\]

let \(\text{path} = \{v_0, \ldots, v_k\}\), then \(\exists i\), such that \(\text{node} = \text{path}.v_i\)

if (there’s stored \(\text{current}\_\text{cost}\))

set \(\text{current}\_\text{cost} = \text{min}(\text{current}\_\text{cost}, \text{stored current}\_\text{cost})\)

if (all the \(\text{node’s parents}\) that should send a message back sent a message back)

send message \(<\text{current}\_\text{cost}, \text{path}>\) to \(\text{node} \text{path}.v_{i-1}\) (\(i > 0\)), or \(\text{node} (i == 0)\)

else

store \(\text{current}\_\text{cost}\)

Figure 3: The downward process

The two phases of Local and Global are identical: compute the cost of the entire path (the maximum of costs of its components utilizing one of the cost functions), then select the least expensive path among the candidate paths as the designated for object retrieval. The difference is in the way they decide on stopping the upward phase and starting the downward phase. Local stops after finding the first source in the path capable of delivering the object, where Global stops when it finds that the cost could not be improved by continuing to go upward. Global stops the upward
process when it either detects a saturated link or reaches a root.

Although seems similar to the shortest path problem in graph theory [GJ79], finding the least loaded path in RedHi is different in the way the path cost is computed. The cost of a path here is the maximum cost of all the nodes and links participating in that path rather that the summation of these costs. This makes the traditional shortest path algorithms unapplicable in this case.

The complexity of the upward and downward phases of these heuristics are $O(n \cdot p)$ where $n$ is the number of levels in the hierarchy and $p$ is the number of candidate paths from head-end to roots. Since $n$ is not expected to be high and since the communication cost forces $p$ to be low (Sec. 3.3), the complexity of these heuristics is not high.

5.3 Resource Reservation Policies

When a node is exercising its option to find the best possible path for obtaining an object, it might reserve all the resources along all the paths so that other requests do not grab the resources (a pessimistic approach). Alternatively, it might not reserve the resources hoping that the selected path will not be occupied by other requests (optimistic approach). In this section, we study the advantages and disadvantages of each of these approaches and eventually propose a third approach, termed Server-Lock strategy.

The resource reservation issue can be best shown by an example. Suppose a request $r_1$ arrives at head-end node $A$ for object $O_i$ with 2 Mb/s display bandwidth. Assume node $A$ determines two paths $p_1$ and $p_2$ for obtaining $O_i$. Subsequently, $A$ selects $p_1$ over $p_2$ due to some heuristic. Now assume during this process another request (say $r_2$) grabs one of the resources (say $B$) of $p_1$ and saturates it. Consequently, $r_1$ must now be rejected while $p_2$ is still ready to support the request. This is how an optimistic resource reservation strategy functions. That is, it does not reserve any resources during the execution of heuristic, hoping that no other request needing a resource of either $p_1$ or $p_2$ will arrive until the best path (i.e., $p_1$) is selected. Alternatively, a pessimistic strategy can reserve 2 Mb/s of all the resources of both $p_1$ and $p_2$ during the execution of the heuristic. In
this case, no other request can saturate any resource of $p_1$ because at least 2 Mb/s is reserved. The disadvantage is that a request needing a resource of $p_2$ might now get rejected while $p_2$ will never be utilized by $r_1$.

To avoid the disadvantages of both pessimistic and optimistic strategies, the Server-Lock strategy is proposed. With this strategy, the server exclusively process one request at a time (in sequence), until either a path is chosen or the request is rejected. Only then the server can process the next request. To observe, with our previous example, suppose node $B$ in $p_1$ receives a message from $A$ asking for 2 Mb/s. Now $B$ will propagate the message up in the hierarchy. Meanwhile, a message generated by $r_2$ reaches $B$. This message will be queued (and not granted as in the case of optimistic) until $r_1$ is completed. Note that during the processing of $r_1$, Server-Lock will not reject any request needing a resource of $p_2$ (as in the case of pessimistic). Instead, it will queue the generated message at the bottleneck node until $r_1$ releases $p_2$ due to its selection of $p_1$. At that time the bottleneck node will be dedicated to the new request.

6 Adaptive Buffering Mechanisms

Until now, we assumed a restrict pipelined object delivery. That is, a request $r$ referencing a 2 hour MPEG-II movie with $R_c = 2$ Mb/s display bandwidth requirement will get accepted only if the system can reserve exactly $R_c$ through a path. However, using memory (or disk) buffers at the client side, we might be able to relax this assumption. To illustrate, let’s denote $R_p$ as the minimum available bandwidth through the path. Trivially, $R_p$ determines the total available bandwidth of the path. Assuming $R_p < R_c$, our restrict pipelined delivery will reject $r$. However, if we buffer a portion of the object at the client side prior to its display initiation, we would be able to guarantee uninterrupted delivery of the object by utilizing only $R_p$ (Lazy Pipelining). On the other hand, if $R_p > R_c$ we can deliver the object faster by utilizing $R_p$. In this case, since the display is restricted by $R_c$, we still need some buffer space at the client side to store the portion which is delivered but not yet displayed (Eager Pipelining). The advantage is that Eager frees up
resources earlier. For the rest of this section, we describe each of these methods in more details. Note that restrict pipelining is a special case of either Eager or Lazy where \( R_p = R_c \).

First, we start by describing a general pipelining mechanism which is based on \( [GS94, GDS95]^2 \). Suppose an object \( X \) is split into \( n \) fixed size blocks \( (B_1, B_2, ..., B_n) \). A block represents a contiguous portion of an object and determines the unit of transfer from the server to a client (e.g., a fixed packet size). Briefly, the pipelining mechanism groups the blocks of object \( X \) into \( s \) logical slices \( (S_{X,1}, S_{X,2}, S_{X,3}, ..., S_{X,s}) \) such that the display time of \( S_{X,1} \) \( (T_{Display}(S_{X,1})) \) eclipses the time required to transfer \( S_{X,2} \) \( (T_{Transfer}(S_{X,2})) \), \( T_{Display}(S_{X,2}) \) eclipses \( T_{Transfer}(S_{X,3}) \), etc. This ensures a continuous display. The time required to transfer an object \( (T_{Transfer}) \) depends on the available resources (server and link bandwidth). This bandwidth remains fixed for the entire object transfer. The following algorithm illustrates this pipelining mechanism.

1: Transfer the block(s) that constitute \( S_{X,1} \) to the client buffer.

2: For \( i = 2 \) to \( s \) do

   a. Initiate the transfer of \( S_{X,i} \) from server to the client buffer.

   b. Initiate the display of \( S_{X,i-1} \).

3: Display the last slice \( (S_{X,s}) \).

The duration of Step 1 determines the latency time observed by the request. During Step 2, while the subsequent slices are transferred from the server, the client buffer resident slices are displayed. Step 2.a and 2.b correspond to two different processes that execute in parallel. Step 3 displays the last slice transferred to the client.

The amount of required buffer at the client side as well as the latency time depend on the production rate \( R_p \) (the rate the object is read from the server and transferred over the network) and the consumption rate \( R_c \) (display bandwidth). The detailed calculation to compute these values are provided in \( [GS94] \). Here, we assume that \( R_p \) is a function of the system congestion.

---

\(^2\)In \( [GS94, GDS95] \), the pipelining was introduced between the tertiary and secondary storage device.
<table>
<thead>
<tr>
<th>Level</th>
<th>Vertices ($V_{\text{level}}$)</th>
<th>Servers</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MaxB(Mb/s)</td>
<td>MaxC(GB)</td>
</tr>
<tr>
<td>1</td>
<td>{0}</td>
<td>384</td>
<td>583.2</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
<td>96</td>
<td>194.4</td>
</tr>
<tr>
<td>3</td>
<td>{4,...,12}</td>
<td>48</td>
<td>64.8</td>
</tr>
<tr>
<td>4</td>
<td>{13,...,39}</td>
<td>24</td>
<td>21.6</td>
</tr>
</tbody>
</table>

Table 1: Pure Hierarchy Parameters

Moreover, we assume that $R_c$ is fixed. The size of the first slice $S_{X,1}$ is equal to $B_1$ when $R_p \geq R_c$. The calculation of $S_{X,1}$ when $R_p < R_c$ is also a function of both $R_p$ and $R_c$ (see [GS94]).

With pipelining approaches, we assumed that the data is buffered at the user site. These approaches can, however, be generalized by allowing any intermediate server node along the path from the server to the user to buffer the data. Moreover, it is important to point out that these buffering techniques are independent of the caching techniques discussed in Sec. 4.

We described Eager and Lazy pipelining only as mechanisms. A policy is required to decide when the system should reject a request as opposed to accept it utilizing $R_p < R_c$. Similarly, when $R_p > R_c$, a policy is required to determine how much of $R_p$ should be utilized for serving the request. We cannot evaluate the mechanisms without having the policies. Design of these policies as well as their performance evaluation are part of our future research.

7 Performance Evaluation

RedHi is a complex network that is difficult to analyze theoretically. Therefore, we decided to conduct a number of simulation experiments to obtain some insights about it. Our main objective is to compare RedHi to a pure hierarchy with respect to fault-tolerance and load balancing. We also compared the performance of our different replacement policies and heuristics.

7.1 Simulation Model

In our simulation, the Distributed Server Architecture $DSA$ has 4 levels. Similarly pure hier-
<table>
<thead>
<tr>
<th>Level</th>
<th>Vertices ($V_{level}$)</th>
<th>Servers</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MaxB(Mb/s)</td>
<td>MaxC(GB)</td>
</tr>
<tr>
<td>1</td>
<td>{0.40}</td>
<td>192</td>
<td>583.2</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
<td>96</td>
<td>194.4</td>
</tr>
<tr>
<td>3</td>
<td>{4,..,12}</td>
<td>48</td>
<td>64.8</td>
</tr>
<tr>
<td>4</td>
<td>{13,..,39}</td>
<td>24</td>
<td>21.6</td>
</tr>
</tbody>
</table>

Table 2: RedHi Parameters

archy is assumed to be a 4-level 3-ary tree. The parameters of these hierarchies are detailed in the Tab. 1 and Tab. 2. It is important to point out that the total amount of resources (server bandwidth and capacity as well as links bandwidth) assigned to these two architectures were identical. For example, although with RedHi the number of links is twice than that of pure hierarchy, the capacity of each link of RedHi is half of the corresponding link capacity of pure hierarchy.

A static movie population was assumed: all movies were initially stored at the root node(s), and no new movie has been inserted during the execution of the simulation. Each root node contains a total of 324 movies.

For each simulation experiment, 18000 user requests were generated. Poisson distribution was used to simulate the inter-arrival period of requests per each head-end. To simulate movie selections, Zipf's law [NPSS95] was employed. All the movies were assumed to have identical lengths and bandwidth requirements. Without loss of generality, each movie is a 2 hour MPEG-2 compressed with a 2 Mb/s bandwidth requirements.

### 7.2 Simulation Results

A request only arrives at the head-end nodes. Subsequently, a request is either served immediately, or rejected if its required resources cannot be found. The number of rejected requests was used as the performance measure. Each experiment was repeated multiple times with different seeds for

---

3We ignored the time required to select and reserve a path as well as the time elapsed until the first block of the requested object reaches the head-end node. We expect that due to fast speed of network links, this delay is in order of seconds. We plan to quantify this as part of our future research.
our random number generators. The system behavior was recorded under variable system loads with all head-end nodes receiving identical arrival rate for requests using Poison. Hence, for all the reported graphs, the x-axis is the percentage of the load on each head-end (varied from 40% up to 240%), and the y-axis is the number of rejected requests. Each experiment had a warmup period for 2000 requests during which the objects were migrated/replicated from the root(s) to other levels of the hierarchy. The results are only reported for the post-warmup period.

The following four sets of simulation results will be detailed: 1) Architecture: Pure Hierarchy vs. RedHi, 2) Cost Functions: FreeBW, RatioBW and UserBW, 3) Object Locating Method: Local vs. Global, 4) Replacement Policy: LRU vs. LFU. Unless stated otherwise, the default parameters for all of these simulation experiments were: Optimistic (Resource Reservation Policy), FreeBW (cost function), LFU (replacement policy), Local (Object Locating Method).

7.2.1 Pure Hierarchy vs. RedHi

In normal mode of operation, fewer requests were rejected by RedHi than by the pure hierarchy (see Fig. 4a). Trivially, under low system load ($\leq 120\%$), the number of rejected requests is very low with both architectures. Under higher load ($> 120\%$), however, the superiority of RedHi is obvious.
For example, when the system load was 200%, pure hierarchy rejected 168% more requests than RedHi. The load balancing capability of RedHi enabled it to function much better under higher system load.

To simulate a link failure, we disabled a link (selected at random) connecting a head-end to a one level higher node after serving 9000 requests. The impact of this failure was analyzed by comparing the performance of the system with and without the failure. Fig. 4a shows that RedHi performed almost identically with and without failure. The performance of pure hierarchy, however, degraded significantly due to the link failure. An interesting observation is that RedHi did not reject any request when \( \text{load} \leq 100\% \) while pure hierarchy rejected 2.5% of the 9000 requests arrived after the link failure. We also compared RedHi with Pure hierarchy in the presence of two failures. In the presence of two link failures (not shown in the graph) the performance of RedHi still remained almost unchanged, while with \( \text{load} \leq 100\% \) pure hierarchy rejected 5% of the 9000 requests.

### 7.2.2 Cost Functions

Cost functions are important parts of our object locating and retrieval heuristics. The three cost functions presented in Sec. 5 were compared. Fig. 4b shows that \( \text{RatioBW} \) and \( \text{UserBW} \) consistently outperformed \( \text{FreeBW} \). For example, when \( \text{load} = 200\% \), \( \text{RatioBW} \) rejected 21% and 80% less requests as compared to \( \text{UserBW} \) and \( \text{FreeBW} \), respectively. As mentioned earlier, \( \text{FreeBW} \) has a tendency to retrieve objects from the higher levels of the hierarchy. Therefore, more links become occupied to retrieve these objects resulting in bad load balancing.

### 7.2.3 Object Locating Methods: Local vs. Global

Fig. 5a depicts the comparison between Global and Local. We expected to see a better performance for Global since it attempts to find the least expensive path globally. Instead, our simulation results show the inverse to be true. This is because Global by selecting more higher level (closer to \( \text{root} \)) nodes to deliver objects occupies more links (similar to the problem observed by \( \text{FreeBW} \)).
This means that the system will use more resources to deliver the same number of objects. Although these resources were less loaded (according to the cost function) at the time that the request arrived, using more resources to deliver objects in the long run yields more rejections. As a result, Local outperformed Global (e.g., 47% less rejection with Local as compared to Global when load = 180%).

7.2.4 Replacement Policy: LRU & LFU

Finally, we compared the performance of the replacement policies. A large margin of difference between the performance of LRU and LFU was observed (see Fig. 5b). This is because LFU is very consistent with the employment of Zipf’s law.

We did not compare the performance of the three resource reservation strategies discussed in Sec. 5.3. We were not able to investigate the performance of these strategies due to the fact that we assumed a zero message delay in our simulation model. Currently, we are modifying our simulation model to incorporate these message delays.
8 Conclusions

In this paper, we proposed a redundant hierarchical (RedHi) architecture for distributed continuous media servers. We showed that RedHi has a better fault-tolerance and load balancing capabilities. With RedHi, the task of locating an object and the decision to where to retrieve it from are performed in a distributed manner. Distributed object management eliminates the possibility of bottlenecks and makes the system more fault-tolerant.

We conducted a set of experiments to illustrate RedHi’s fault-tolerance and load balancing capabilities. We observed that under the same load conditions, RedHi can serve more requests as compared to the pure hierarchy. We also observed that a link failure degraded the performance of the pure hierarchy while having almost no effect on the performance of RedHi.

Other experiments were performed to gain some insights about RedHi. When comparing object locating methods, it was observed that Local consistently outperformed Global. Moreover, LFU replacement policy was found to be superior to LRU. Out of the three cost functions presented in this paper, RatioBW showed the best performance while FreeBW was the worst.

This study can be extended in many ways. As short term plans, we intend to consider the network delay. Consequently, we can compare the impact of our resource reservation strategies (Sequential, Optimistic, and Pessimistic). Once network delay is introduced, we also intend to add a new cost function which takes into the account the length of a path for object retrieval. We also plan to design and evaluate policies to study the impact of employing the adaptive buffering mechanisms on the system performance. Finally, we intend to investigate partial object replication and migration as well as dynamic request migration. As a long term plan, we want to investigate new challenges introduced by other applications when assuming RedHi. For example, the temporal relationships introduced by digital editing applications [CGS95] or query scripts introduced by MM-DBMS [SDG97] applications.
References


