2006

Cooperative Interval Caching in Clustered Multimedia Servers

Kim Tran
San Jose State University

COOPERATIVE INTERVAL CACHING

IN

CLUSTERED MULTIMEDIA SERVERS

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Kim Tran

December 2006

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

_____

Dr. Suneuy Kim

_____

Dr. Robert Chun

_____

Dr. John Avila

APPROVED FOR THE UNIVERSITY

_____

Dr. Suneuy Kim

_____

Dr. Robert Chun

_____

Dr. John Avila

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

Video On Demand (VOD) is one of many prominent internet services that have a rapidly growing service market. Many popular internet web sites, like Yahoo!, Google, and CNN are now offering VOD to their millions of customers. One of the most popular VOD websites is *youtube.com*, where users can upload and view homemade videos without charge. According to Asia Media [1], *youtube.com* is home to 25 million videos, and streams 15 million of them each day. The playback of video on *youtube.com*, Yahoo!, Google, and similar sites is smooth, and has moderate picture resolution and frame rate. This achievement is the result of a great amount of research to improve the VOD server performance over the last decade. However, it is still considered to be challenging to design servers that have capacity for rapidly growing demands of video services because of the characteristics of video objects: they require continuous and seamless presentation at a specified play back rate (e.g. 1.5 Mbps for MPEG1 movies and 4Mbps for MPEG2 movies), and they have a large data size (e.g. 1GB for 1 hour-long MPEG1 movies).

Video server performance is relying on many different factors including disk bandwidth and capacity, network bandwidth, clustering, and cache management. To improve network bandwidth, multicast protocols [8, 9, 10, 11, 12] and broadcast protocols [2, 4, 6, 7] have been proposed. To reduce latency and increase the number of simultaneous streams, disk partition and data distribution (stripping or replication) are studied in [13, 14, 15]. The client/server model is transformed to clustered VOD servers to improve scalability and eliminate the issue of having a single point of failure [16, 17, 18]. Cache management at both client and server ends is carefully examined, applying techniques like multicast cache [19], peer-to-peer proxy [20], interval caching [21], optimal chaining [22].

Among these numerous approaches to boost server performance, we closely examined caching and server clustering techniques. These techniques can lead to desirable features of video servers encountering rapidly increasing demands: disk bandwidth saving, high scalability and availability.

To choose the best caching technique for VOD, one should note that traditional caching algorithms, such as LRU, do not work well for video servers due to the large size of video objects. When the entire video objects need to be cached, a server cache may not be able to hold more than a few video objects; this results in a low cache hit ratio. One alternative to traditional caching algorithms is interval caching, proposed in [23] and [21]. This is an innovative caching algorithm developed for video servers. It employs temporal locality of client requests to the movies and caches intervals between successive requests to the same movie, as opposed to the entire video object. Interval caching has been adopted in many research works as caching scheme for video servers [24, 25, 26].

Clustering is known to be a leading approach to developing highly available and scalable servers. With high availability, failure of a cluster member will not dramatically degrade the server performance and the remaining cluster members can continue to service clients with reasonable quality of service. Scalability is an essential feature of servers to manage the fast growth of service requests. With all these major advantages, scalable VOD server clusters have been researched to respond to the steeply increasing client requests to VOD services [27, 24, 16, 17, 18].

In this project, we develop a cooperative interval caching (CIC) algorithm for VOD server clusters. With CIC, distributed caches in the cluster cooperate together to accommodate as many requests as one large aggregated cache would do. Cache cooperation in video servers has been studied in [24,25]. However, the cooperative cache schemes proposed in the previous works have either one or both of the following drawbacks.

- Communication overhead: servers in the cluster exchange messages to determine which server will handle the given request.
- Single point of failure: one server is dedicated to forward the given request to the appropriate server. As a result, this dedicated server becomes a possible bottleneck and the single point of failure of the system.

The CIC algorithm we develop in this project resolves these issues. The followings are the beneficial features that the CIC algorithm provides the system with:

- High hit ratio of distributed caches in a server cluster; this would be close to that of a server with a single aggregated cache.
- High availability achieved by propagating the responsibility of a failing server to another without significant server down time. There will be no single point of failure, however, since all servers within the cluster are capable of performing identical tasks.
- Low communication overhead is required to support cache cooperation because a hash function is used to find the server for the given request.
- High scalability is achieved by using a mechanism to add/remove servers to/from the cluster with little performance degradation.

The performance of CIC is evaluated through simulation. In the performance study, we investigate how CIC improves the capacity of the clustered server, which is defined as the number of concurrent video streams that the server cluster can accommodate while satisfying the quality of services at client sites. Without caching, the capacity would be determined by the overall bandwidth of the disk system in the cluster. Caching at individual server in the cluster will improve the capacity by servicing more streams from the cache – streams that would otherwise be rejected. CIC further increases the number of cached streams in the cluster by having caches cooperate together. With a given disk capacity, the capacity of a clustered server is determined by how many concurrent streams the caching system can accommodate. Therefore, we chose the *average number of cached streams* as a performance metric. The performance gain from cache cooperation should be achieved with a reasonable amount of inter-cache communication overhead. To estimate the required communication overhead to support cache cooperation, we measure the number of more operations required by cooperative interval caching, as compared to the non-cooperative version. The performance of CIC is measured over various system and workload parameters including cache size, arrival rate, and movie access pattern. The impact of cache cooperation on server performance is studied. We also compare different approaches to select a server in the cluster that will serve a given request from its cache.

The following list summarizes our findings from the performance study:

- The server cluster with CIC can support twice as many cached streams (95%) as the server cluster with non-cooperative interval caching.
- It has been known that with interval caching the number of cached streams is increased as cache size increases, but the increment is not proportional to the increment in the cache size. That is because as the cache size increases, the number of cached streams is increased, but the possibility of getting larger intervals is also increased. We confirm that this feature of interval caching is still preserved for cooperative interval caching.
- With CIC, two major processes should be done to find available cache for a given request: to find a server maintaining information about the preceding request of the given request and to find another server when the current server turns out not to have enough cache space for the given request. We found that it is better assigning the requests to the same movie to the same server in a way that the server maintains all the information about these request. Basically, this mechanism will help a request to find information about its preceding request without much of communication overhead. We also found that if the current server doesn't have available cache space, random selection works well to find the next server with available cache space.
- With CIC, the average number of cached streams in the clustered environment is about the same or greater than that of a single server environment with the same total cache size. In an integrated environment where there is a single cache, all requests are sent to one server, and cache replacement evicts the largest cached interval to support the newly arrived and smaller cached stream. In the cluster environment, each server has a different maximum cached size stream, which changes the cache replacement values, and the result is that in some cases the number of total cached streams is bigger.

The rest of this paper is organized into seven more sections. Section 2 briefly introduces some related works. Section 3 provides readers with detailed information of interval caching. Section 4 illustrates CIC system model. Section 5 explains CIC algorithm.

Section 6 is about the simulator. Section 7 is for performance evaluation. Section 8 summarizes and discusses future works.

2 Related Works

In this section, we study research related to caching in stand-alone as well as in clustered multimedia servers. Then we study the cooperative interval caching in a clustered media server.

2.1 Caching in stand alone multimedia server

Caching in a multimedia server is different from other types of caching. Since video size is large, caching the whole item is not efficient. A one-hour MPEG-2 video with a playback rate of 4 Mbps requires about 1.8 GB of memory. Therefore, a cache memory cannot accommodate more than a few video objects, which results in a low cache hit ratio. If we choose to cache a video partially, the caching algorithm should ensure that the video data is delivered to the users seamlessly.

Subhabrata Sen et al. proposed a prefix caching technique whereby a proxy stores the initial frames of popular clips. Upon receiving a request for the stream, the proxy "initiates transmission to the client and simultaneously requests the remaining frames from the server" [28]. This scheme provides quick responses to popular requests, but it sacrifices ones of low demand.

Kun-Lung Wu et al. proposed the segment base proxy caching of multimedia streams technique to handle the delayed start issue mentioned above. Their paper goes one step farther to handle a cached media object that was once hot but has recently turned cold. Instead of caching fixed size segmentation, blocks of a media object received by the proxy server are "grouped into variable-sized, distance-sensitive segments" [31]. The segment size increases exponentially from the beginning segment. If a cached object that was once hot but has since turned cold is detected, all its related cached segments can be quickly discarded in big chunks.

Asit Dan and Dinkar Sitaram proposed an innovative way of caching video data called "interval caching" in [23]. Interval caching exploits the temporal locality of requests to the same movie to determine which portion of movies can be and should be cached to

maximize the cache utilization of the cache and also to ensure continuous delivery of data to the users. If there is no space for the new interval and if there are larger cached intervals than the new one, the largest cached interval will be replaced with the new one**.**

2.2 Caching in clustered servers

A cluster is a set of servers that work together, using additional network and software technology to leverage all the resources, either for high availability or high performance. There are two main types of clusters: loosely coupled clusters and tightly coupled clusters. In a tightly coupled cluster, all machines have access to the system code in common memory, and all share storage and the file system. In a loosely coupled cluster, each machine has its own file system, storage, and IO subsystem, and each is connected to the others through a high speed network connection. To create a loosely coupled cluster in which all components work together efficiently like a single machine, there are many software and hardware issues to resolve. The essential hardware required to set up cluster are the servers themselves, and a high speed private LAN network to help machines communicate. The cluster software is now available widely with different products, including Sun Cluster, IBM eServer Cluster, HPC - Linux Cluster, and Microsoft Cluster software. We concentrate more on loosely coupled cluster, which has good price/performance ratio. It is much less expensive to buy several Commodity Off The Shelf (COTS) servers than it is to buy one large server that has the same total memory.

There are several research works that studied caching schemes in loosely couple server clusters. Cherkasova, Ludmila and Magnus Karlsson propose a strategy named WARD (Workload-Aware Request Distribution), a caching strategy of the web cluster that aims to “maximize the number of requests served from the total cluster memory” and to “minimize the forwarding by identifying the subset of core files to be processed on any node” [42]. In other words, WARD assigns a small set of the most frequent files, called “core”, to be cached in all servers. The rest of the frequent files are cached by different cluster nodes. The core size is determined through “ward-analysis”, which takes into account the system and workload parameters, such as workload access patterns, number of nodes, memory size, and disk access overhead.

Ethendranath Bommaiah et al. propose a proxy-buffering caching scheme using a cooperative set of helpers, which are "caching and data forwarding proxies inside enterprise networks or ISP networks" [29]. Each server in the cluster will be associated with a helper, which serves all the client's requests to the server. Upon receiving a request, the helper first attempts to serve it. If it does not have enough resources to serve it, it tries to cooperate with other helpers in the cluster. The following excerpt from Ethendranath Bommaiah et al. paper clearly explains how helpers work. "Whenever a request for a specific streaming object is received at a helper for the first time, the helper forwards the request to the server. Upon receipt of the request, the server starts streaming the object to the helper. The helper stores data in its memory and disk, and at the same time streams data to the client." [29]

Internet Cache Protocol (ICP) and CARP (Cache Array Routing Protocol) are proposed in [38] and [39], respectively. Using these protocols, the proxies in the web cluster can determine who can serve the current request. To achieve this goal, the proxy that receives the request sends messages among siblings and parents in the proxy hierarchy, and then waits for the quickest response. The proxy who sent the request first will be the one who serves the request. CARP is an improvement over ICP. Instead of broadcasting messages, it uses a hashing algorithm to determine the route to send requests, thereby reducing communication overhead. In our project, we adopt a similar approach to CARP by using a hash function to find the appropriate server for a given request.

2.3 Interval caching in clustered multimedia servers

Several research works use interval caching in multimedia server clusters. Z. Ge, P. Ji, and P. Shenoy [24] propose a streaming media server cluster architecture called DALA. DALA forms *n* groups, one for each video object it serves, and dynamically allocates servers to each group, adapting to the changing demands on each object. Servers exchange messages in the process of joining/disjoining a group, resulting in communication overhead.

Te-Chou Su et al. [25] propose a client-side caching scheme, called Optimal Chaining. With optimal chaining, each client system adopts interval caching; it cooperates with other client systems to overcome their limited cache capacity and to reduce the use of server streams. That is, if a client cache is too small to hold the interval between consecutive requests, the caches of other clients are exploited to hold the rest of the interval. Similar caching schemes are proposed in [40, 41]. The major different between these and our work is that they are client-side caching schemes using client buffers, and ours is a server-side caching scheme.

3 Interval Caching

In this project, we develop a cooperative interval caching (CIC) algorithm for VOD server clusters. Since the foundation of our work is interval caching, originally proposed in [21], we reserve this section to explain what it is and how it works.

Caching is a term given to storing a certain type of data in faster-to-access resources for later use. Traditional caching policies keep data objects with higher temporal or spatial locality in the cache memory. For example, LRU or LFU exploits temporal locality of data objects and replaces the objects that have been least recently or least frequently used from the cache, thereby enhancing cache hit ratio. These policies are based on the hypothesis that the current request is a good indicator of the future request, and have been successfully used in various areas such as paging and database management. However, they cannot be directly applied to cache video objects due to the size and the continuous playback requirements of video data. Generally the size of video object is large. A one hour MPEG-2 video with a playback rate 4 Mbps requires about 1.8 GB of memory. Therefore, a cache memory cannot accommodate more than a few video objects, resulting low cache hit ratio. Caching the part of the video objects to increase cache hit ratio requires careful management, otherwise the continuous playback requirements will not be satisfied.

In a video server, multiple streams are concurrently getting different parts of video objects depending on which parts of the data are delivered to the client sites. Without user

interactivity through VCR operations, each stream gets video data starting from the beginning to the end of the video object at a specified rate. Streams can simultaneously access the same movie, and will form a sequence, as shown in the Figure 3.1, moving towards the end of the video object while data delivery is progressing. With interval caching, intervals between two successive streams on the same movie are cached. Two successive streams can be called the preceding stream and the following stream. As shown in Figure 3.1, suppose S1 is the stream started reading video data at time *t1*. S2 and S3 have subsequently arrived at the video server at time t2 and t3 respectively; they are accessing the same movie. With this scenario, (S1, S2) forms a pair where S1 is the preceding stream and S2 is the following stream. Also, (S2, S3) forms another pair where S2 is the preceding stream and S3 is the following stream.

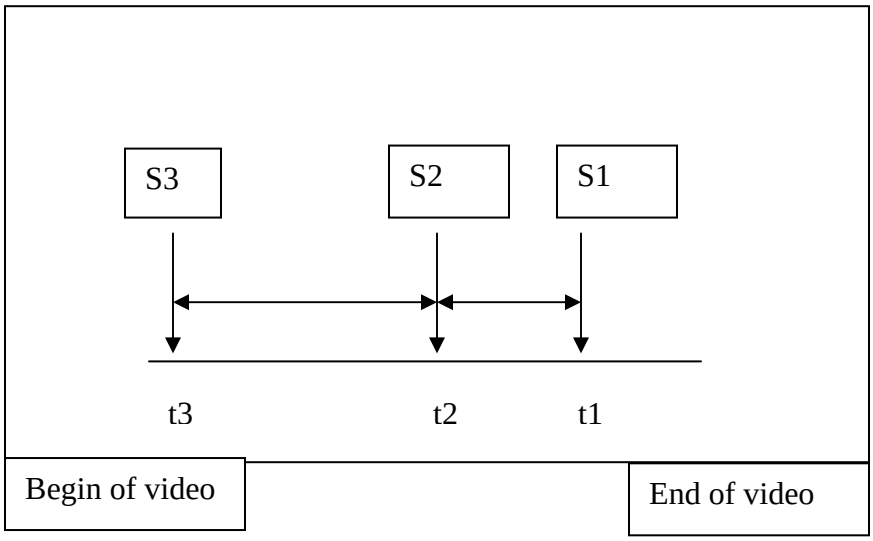


Figure 3.1: Interval Caching

The time interval between two successive streams will be determined by how frequently streams arrive at the movie. It is a function of the arrival rate of requests to the server and of the movie's popularity. The time interval between two successive streams can be translated into the required amount of cache memory to store the interval. For example, with a playback rate 1.5 Mbps, if the time interval between S1 and S2 is 10 seconds, the required cache space for the interval is about 2 MB.

If there is no preceding stream, such as S1, the stream gets data from the disk. As soon as a following stream, S2, arrives at the movie, the interval size between S1 and S2 is calculated. If there is available space in the cache, the interval is cached. If the cache is full and if there exists any interval larger than the newly formed interval, then the larger one will be removed from the cache to release the space to the new one. (This larger interval is called the victim). In this way, interval caching ensures that only the smallest intervals are stored in the cache to accommodate the maximum possible number of intervals.

If there is no available cache space and no victim, the following stream should get the data from the disk. If a cache space is allocated for the interval, the preceding stream S1 starts saving the data it read from the disk in the cache for the following stream S2 to read. After some catching up time, the following stream can get the data from the cache instead of the disk. The preceding stream may be getting data from the cache as for the pair (S2, S3) case. Once the pair is formed and the interval is cached, S2 saves what it used for S3 in their cache space. If both intervals of the pairs (S1, S2) and (S2, S3) are cached, S1 is the only one among three accessing disk and the rest two are getting data from the cache. The following flowchart illustrates the interval caching algorithm:

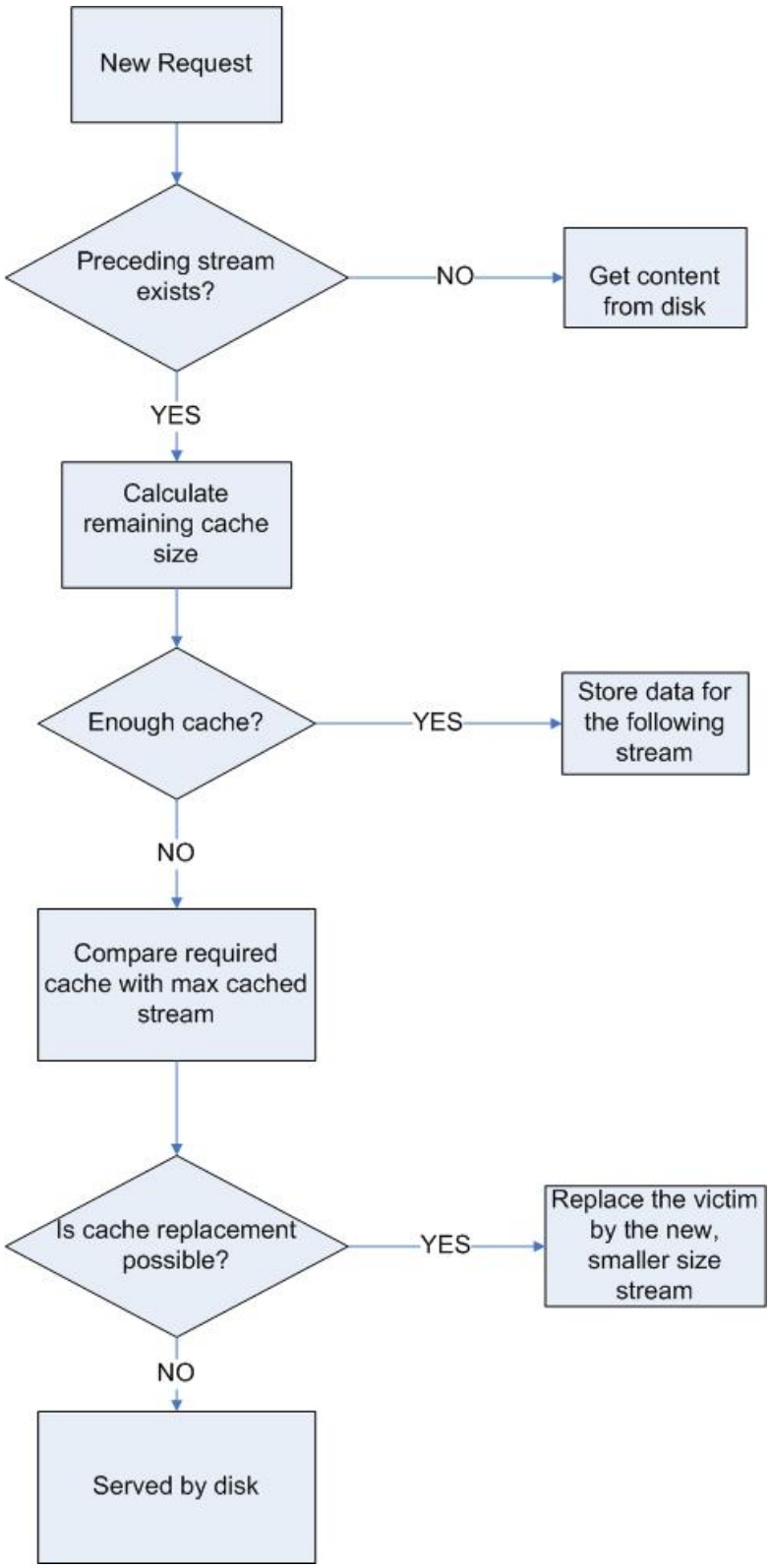


Figure 3.2: Interval Caching Flowchart

4 System Model

Figure 4.1 shows the system model of the cluster architecture, consisting of *n* video servers. Operating systems and hardware platforms of these servers can vary, forming a heterogeneous environment. We can choose any OS (UNIX, Linux, Sun Solaris, etc.), as long as these OS(s) provide TCP/IP software to synchronize processes among servers. These servers communicate with each other through fast connections like fast Ethernet or Gigabit Ethernet, and send video data to the clients via high-speed networks. The video server cluster is assumed to be connected to the secondary storage (like EVA, XP) through a fast I/O interface (SAN switch). We assume that the secondary storage is large enough to hold all the video objects and its interface is fast enough to support a specified number of concurrent streams.

Since VOD servers tend to push large amount of data very quickly, storage area network, SAN, is the best candidate for this job. SAN is an array of disks having intelligent firmware in the controller within the array, which allows SAN do more tasks than what an array of disks can do (caching, generating RAID and parity, creating logical disks, etc). In a SAN, disks are not attached to any specific servers. There is a layer between storage and servers, called the fabric layer. Data is moved through the host, the fabric, and the storage layers at the speed of light. SAN provided two clear advantages to VOD clustered servers. First of all, data is detached from servers. If any of the servers fail, the data storage is intact. Secondly, data is accessible by all servers. This feature increases storage utilization, also improves server performance. Any requests from servers will be sent to SAN once rather than hopping from one server to another to find the right storage. Figure 4.1 gives an overview of SAN's architecture.

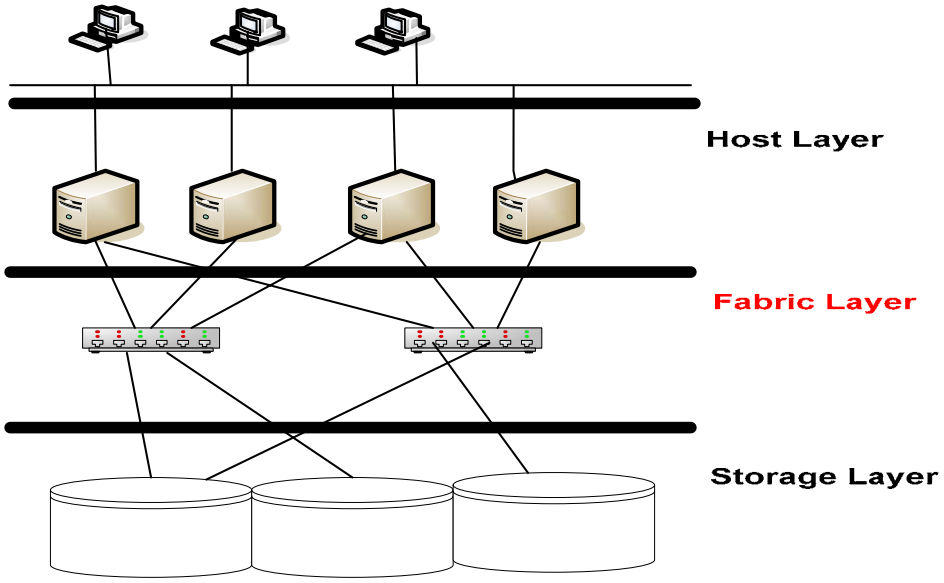


Figure 4.1: SAN architecture

Figure 4.2 illustrates CIC system model. Each server in this model functions identically to support high availability of the clustered video server by removing the possibility of a single point of failure. When a client request arrives at the video server cluster, it is sent to any one of the server in the cluster. This chosen server is responsible to determine the host, which is the best candidate server among servers in the cluster including itself. How a host is selected will be explained in chapter 5. Once chosen, the host examines its local cache to see if it can serve the request. If the local cache cannot service the request, the host forwards the request to another host. This procedure continues until a host is found for the service or until it is determined that there is no such a host. If there is no available host and there is available disk bandwidth, then the secondary storage will serve it (SAN, in this model); if not, the request will be denied. The CIC policy adopts an intelligent method, which will be explained in the next section, to assure that this search chain is reasonably short in length.

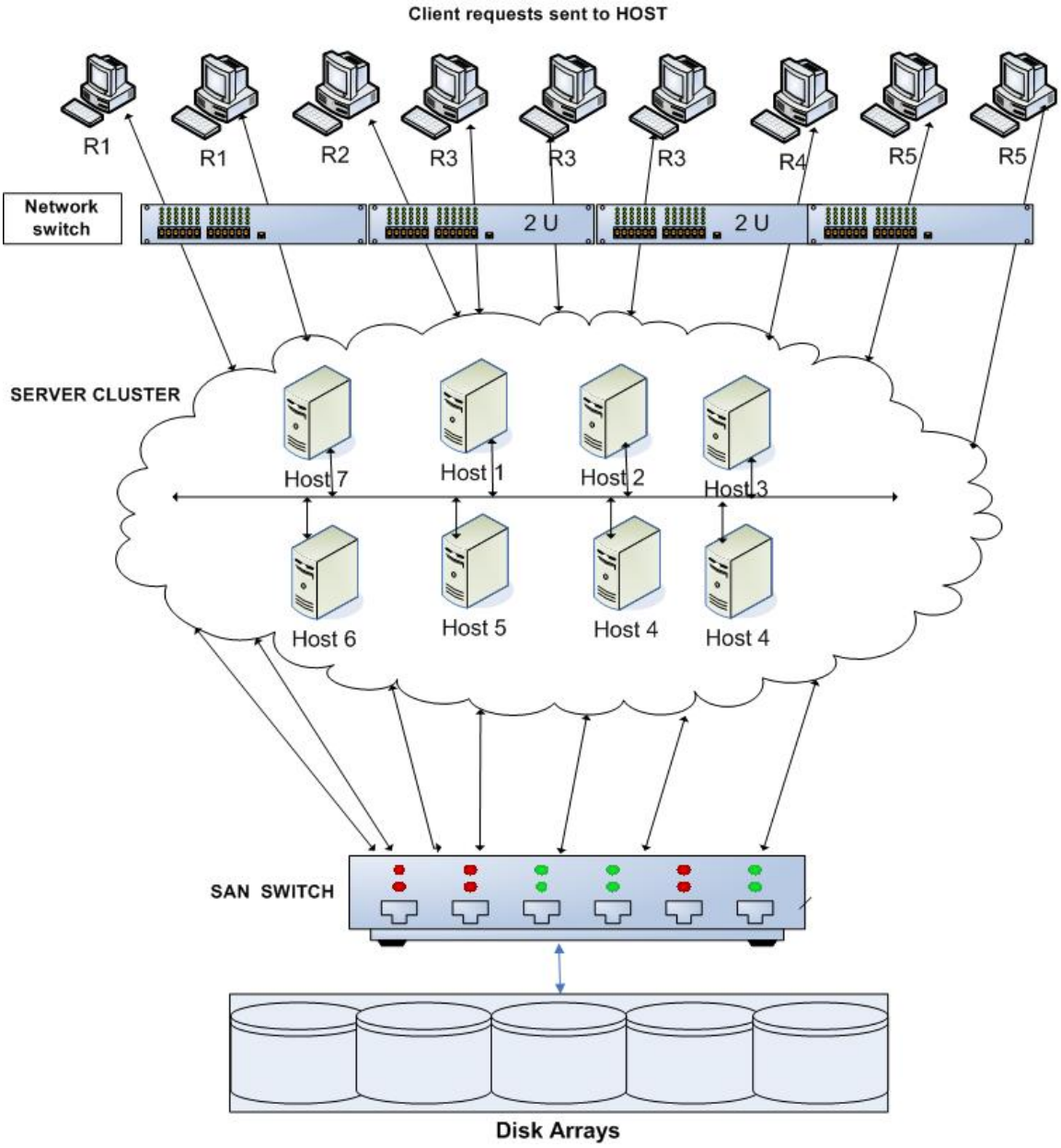


Figure 4.2: CIC system model

5 Cooperative Interval Caching (CIC) Algorithm

The CIC algorithm describes the procedure to find a server in the cluster that can service a newly arrived request from its cache. If no cache in the cluster can accommodate the request, it is directed to the disk system. The request is denied from the clustered server if there is no available disk bandwidth to support the request. The algorithm allows the distributed caches in the cluster to cooperate together to produce a high cache performance close to that of one large cache with the same size as their integral size. The cooperation among caches requires a certain amount of probing to either find a server cache for a given request or to determine the request cannot be serviced from any of the cache server in the cluster. One of the goals of the CIC algorithm is to minimize this probing overhead by intelligently directing a request to the most appropriate server. The CIC algorithm also addresses the scalability issues of a clustered server. With CIC, servers in the cluster take an identical role, thereby removing a single point of failure from the cluster. When a server fails, the cluster recovers smoothly without dramatic degradation of performance. In this section, we first define terminologies that will be used in the CIC algorithm, describe the algorithm, and then explain how the goals of CIC, that is, high cache performance, minimum probing overhead, and scalability, are achieved.

We made the following assumptions in developing the algorithm:

- Users do not generate any VCR operations such as fast-forward, rewind, and pause.
- Once the user starts watching a movie, the user finishes the movie without terminating half way.

Before describing the details of the algorithm, we will define terminologies used in the algorithm.

- ***Host***

The cluster may consist of heterogeneous servers. However, the servers run CIC functioning identically in cache management. We define a server running CIC as host. That is, all servers in the clusters are functionally identical hosts.

- ***Scoreboard and primary host***

A request can arrived at any of the servers in the cluster. When a server receives a movie request, it needs to direct the request to the host with the highest probability of serving its preceding request. Similar approach is used to direct web requests to the proxies in the web cluster in [39]. The server that initially receives the request generates and uses a scoreboard to achieve this purpose. Using scoreboard, all requests for the same movie are directed to the same host to increase the probability of finding their preceding requests.
A scoreboard is a sorted list of all host names in the cluster. How the hosts are sorted will be explained in section 5.1. The host listed on the top of the scoreboard is called the primary host. CIC makes sure to generate the same scoreboard for the requests to the same movie. That means that requests for the same movie will be consistently forwarded to the same primary host

In Figure 5.1, a host generates a scoreboard for each newly arrived request. According to the scoreboards, requests to movie 1 and movie 2 will be forwarded to their primary hosts *star* and *venus*, respectively. Requests to movie 3 are forwarded to their primary host *star*. Note that requests to different movies can be assigned to the same host (*mv1* and *mv3* have the same primary server: *star*). The use of scoreboard can only ensure that the requests to the same movie are always assigned to the same primary server.

Figure 5.1 illustrates how requests are distributed based on scoreboards.

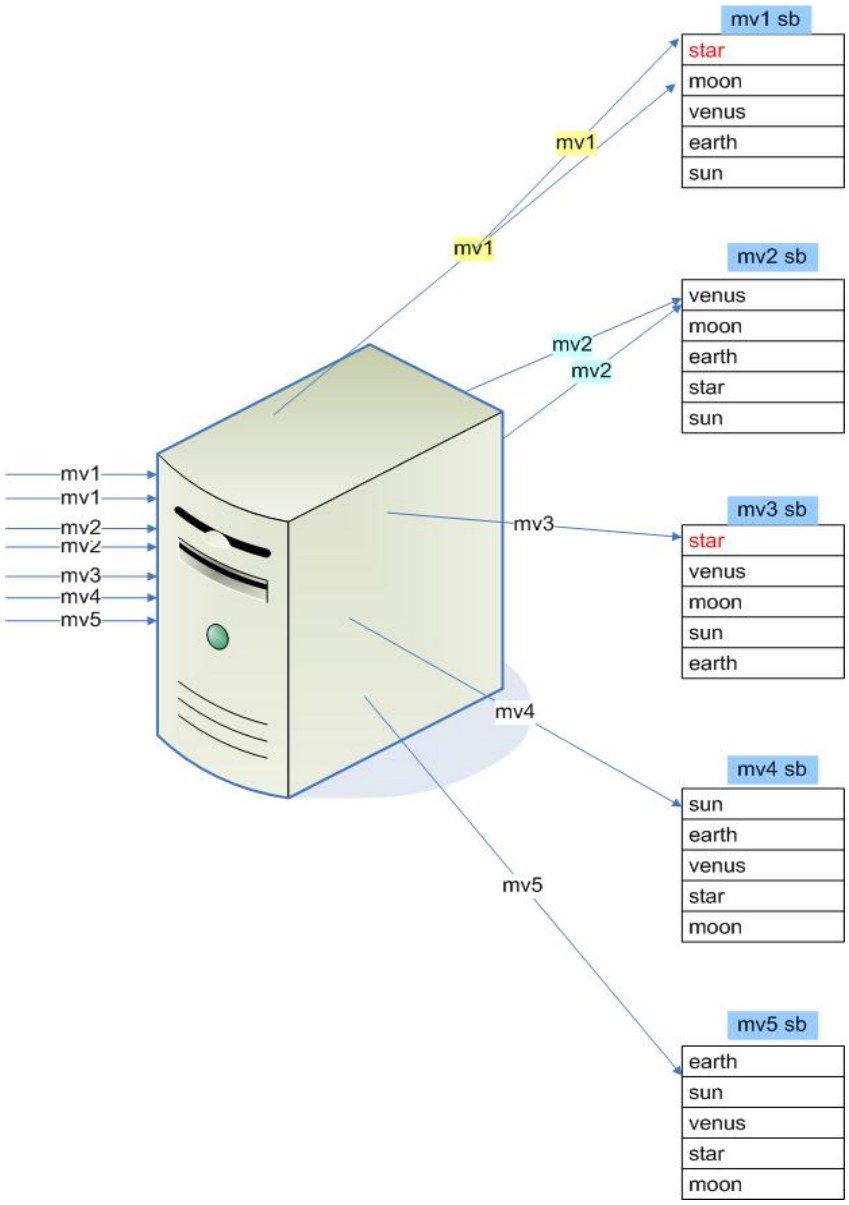
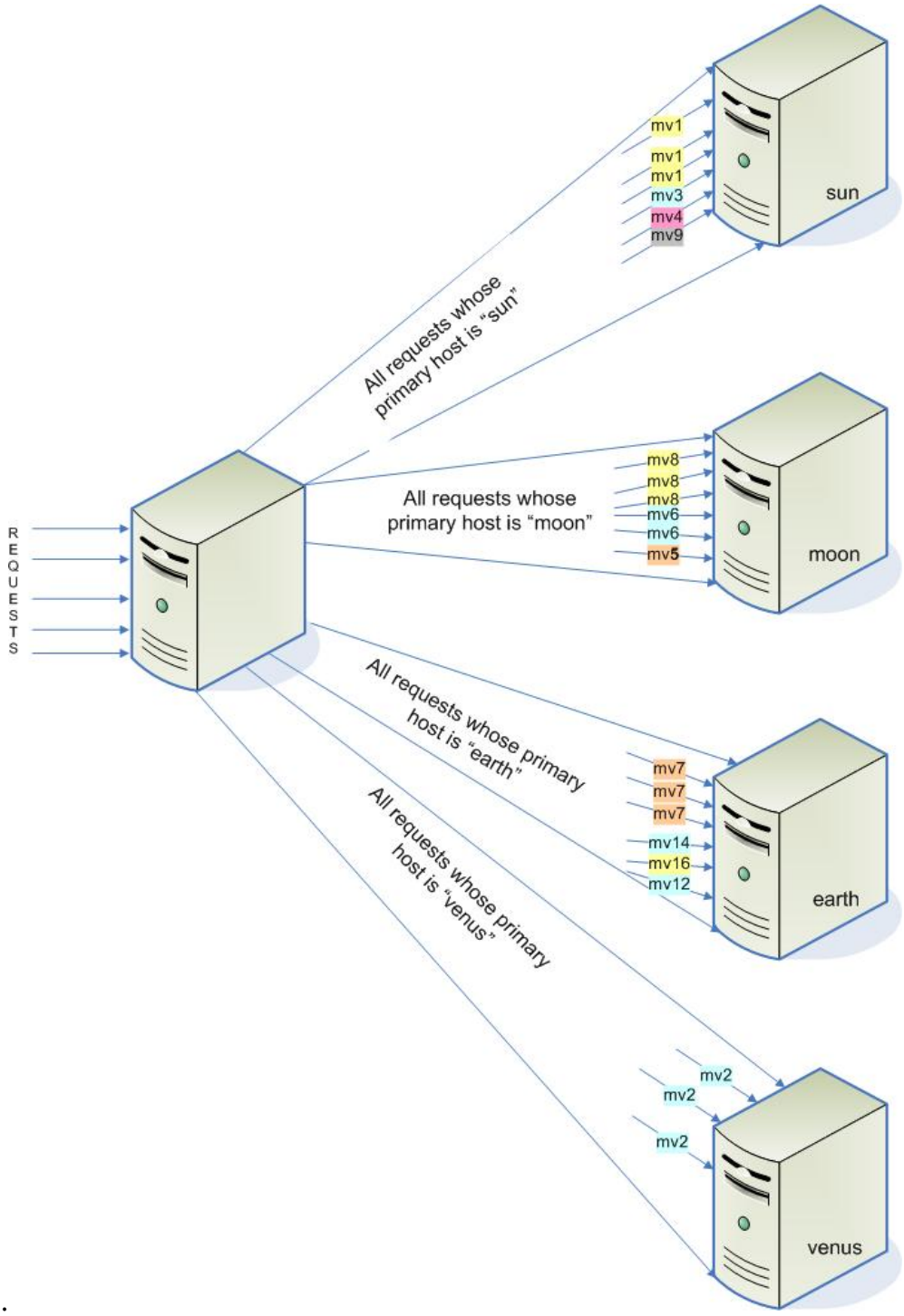


Figure 5.1: request distribution using scoreboard

The Figure 5.2 below shows how requests are grouped and primary servers are assigned to groups.



.

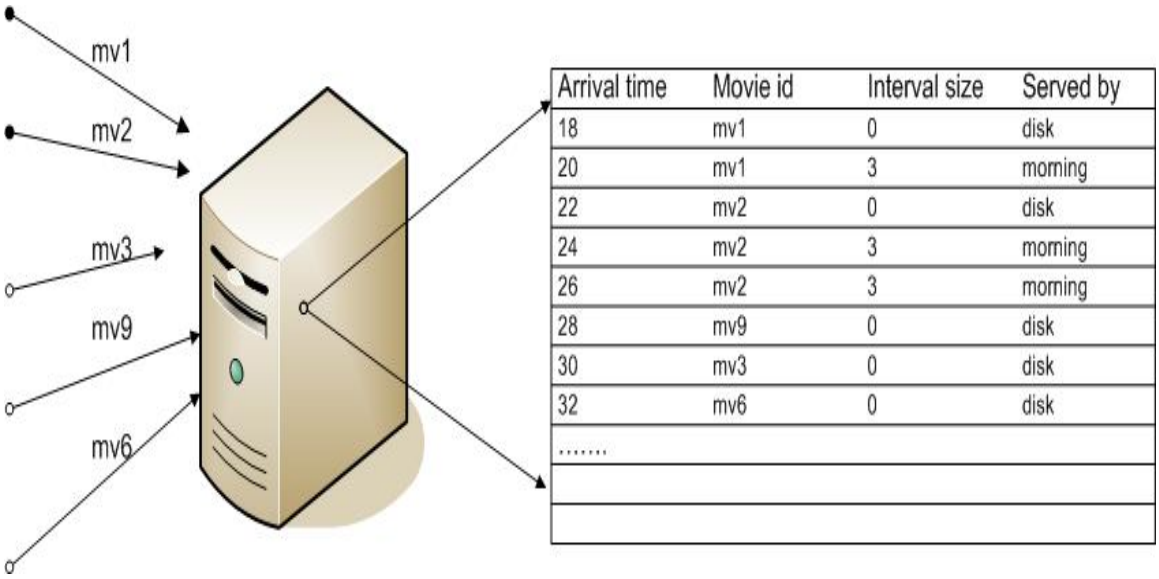
Figure 5.2: requests are grouped; primary servers are assigned to groups

- ***Primary request***

Requests assigned to the primary host. In figure 5.2, mv1 is the primary request of host *sun* and mv8 is the primary request of host *moon*.

- ***Registry***

Registry is a table reflecting the cache's content. It keeps track of all the host's primary requests. Figure 5.3 shows details of a registry. It records movie id, time the movie is requested, and the interval size. If the movie is served by disk, the interval is 0.



Arrival time	Movie id	Interval size	Served by
18	mv1	0	disk
20	mv1	3	morning
22	mv2	0	disk
24	mv2	3	morning
26	mv2	3	morning
28	mv9	0	disk
30	mv3	0	disk
32	mv6	0	disk
.....			

Figure 5.3: registry content

Line #2 of figure 5.3 shows movie *mv1* arriving at time 20. Its preceding stream is *mv1* (line #1), arriving at time 18. In line #2, the interval time is (20-18) * transfer rate = 2 * 1.5 = 3, and it is served by server *morning.* Line #3 of figure 5.3 shows movie *mv9* arriving at time 22 with no preceding stream. This *mv9* is served by disk.

Any stale streams will be removed from the registry. Stale streams are streams staying in the registry a period of time longer than the duration time of the movie. As they are removed, the caches they occupy are returned to the cache pool to serve coming streams which meet the interval caching requirements.

The table 5.1 shows the pseudo code of the CIC algorithm and the rest of the sections in this chapter are dedicated to describing the core of the algorithm.

```
main
      request sent to server
      server creates scoreboard
      request sent to primary host
      remove stale streams from host registry
      insert the new stream to registry
      readState = “00”
      totalRequest++
      No preceding stream
            if host is primary server
                  servedByDisk++
                  readState  = ‘01’
            else if host is a non-primary server
                  compare buffTransfer with cacheCounter
                  if enough cache, do
                        intervalSize = buffTransfer
                        cacheCounter -= buffTransfer
                        servedByCache++
                        updateRegistry
                  else if not enough cache
                        if cacheReplacement succeeds, do
                           intervalSize = buffTransfer
                           cacheCounter+=return (cacheReplacement)
                           servedByDisk++
                           victim’s readState = ‘01’
                           updateRegistry
                        else if cacheReplacement fails
                           nextServer
                           readState = '03'
      Preceding stream exists
            calculate buffRequired
            compare buffRequired with cachCounter
            if buffRequired < cacheCounter
                  intervalSize = buffRequired
                  cacheCounter -= buffRequired
                  servedByCache++
                  updateRegistry
            else if not enough cache
                  if cacheReplacement succeeds
                        intervalSize = buffTransfer
                        cacheCounter += return (cacheReplacement)
                        victim’s readState = ‘01’
                        updateRegistry
                  else if cacheReplacement fails
                        nextServer
                        readState = '03'
end of main
updateRegistry()
begin
     update servedBy
     update readState
end
```

53 **Table 5.1: CIC Algorithm**

5.1 Request Distribution

A new request can arrive at any server in the cluster. The server that initially takes the request produces a scoreboard for the request to determine the primary host, the host with highest probability of serving the preceding request of the new request.

Figure 5.4 illustrates how each request is forwarded to the primary host based on the scoreboard. Suppose requests for *mv1*, *mv2*, *mv3*, *mv4* and *mv5* arrived at a server in the cluster. In this case, the server would generate five scoreboards, one for each request.

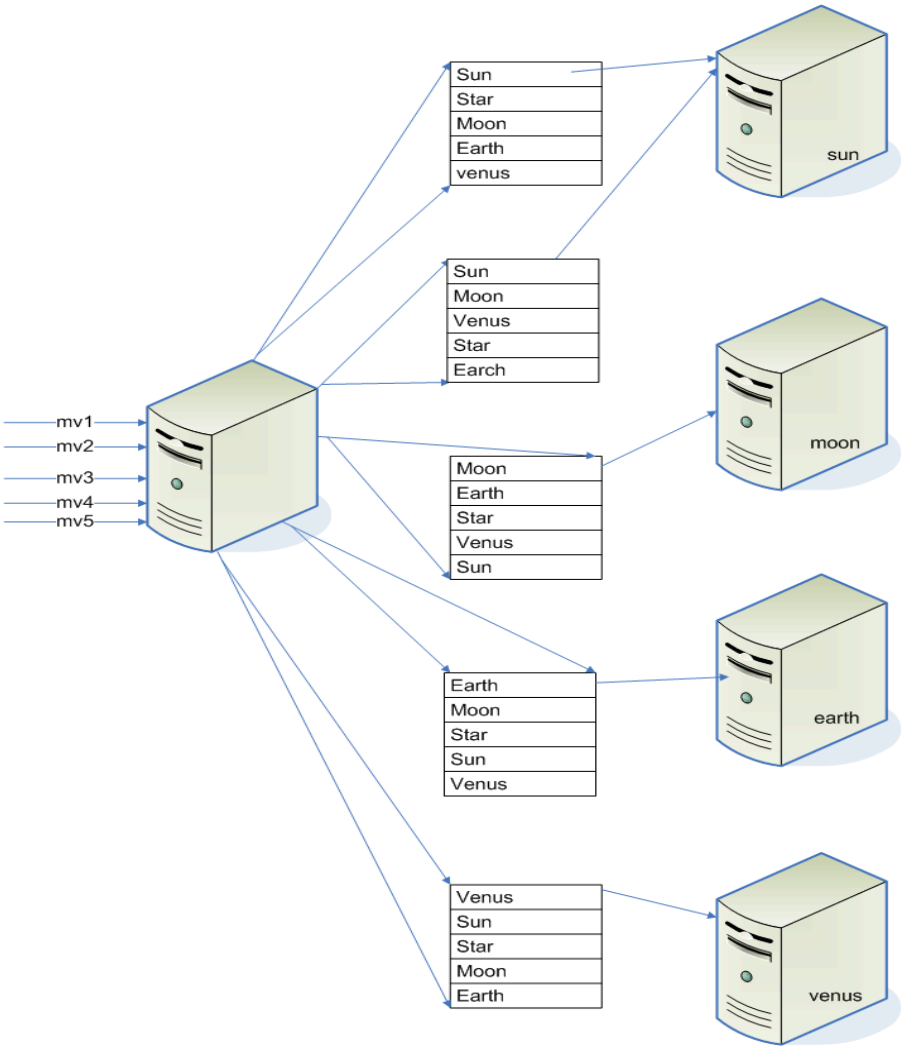


Figure 5.4: various requests distribution

A scoreboard is a sorted list of all hosts of the cluster in the descending order of score. The host at the top of the list is the primary server that the new request is forwarded to. The server's score is based on the server name and the request id. Each score will be calculated as follows:

1. Hash the request id to get a unique *h_id*
2. Hash each server name to get a unique *h_serverName*
3. xor *h_id* with each *h_serverName* in the scoreboard to get a unique pair of server – score
4. Sort the scoreboard, in descending order of scores.

The above calculation ensures that the identical scoreboard is generated for the requests to the same movie. That is, every server in the cluster can forward requests of the same movie to the same primary host as illustrated in figure 5.5. Such algorithm achieves the highest probability of finding the preceding streams, satisfying the locality requirement of interval caching.

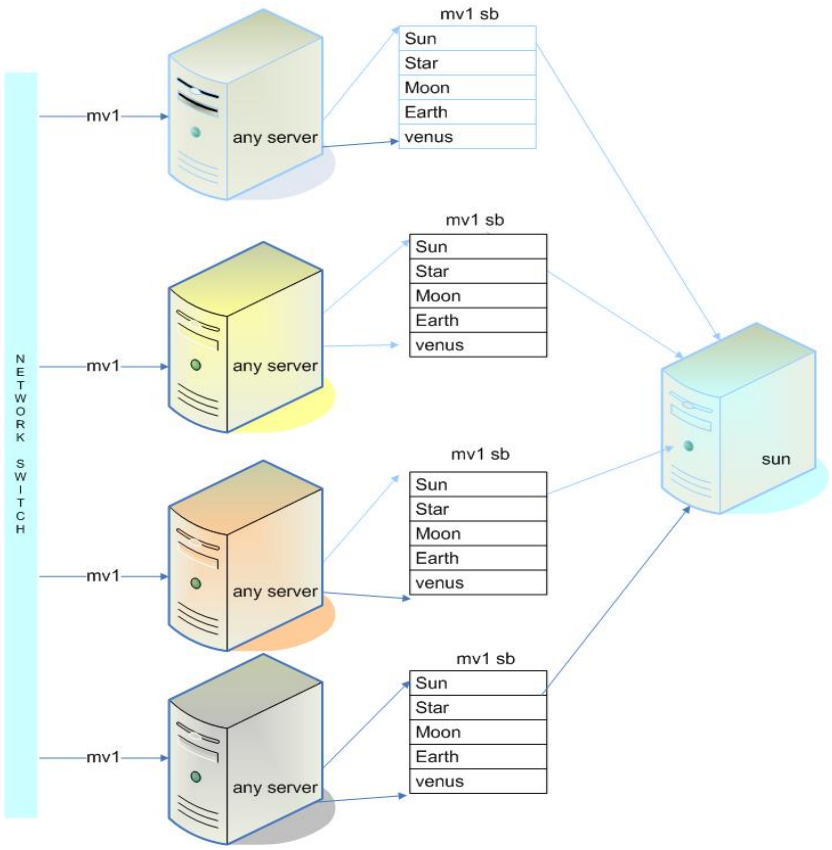


Figure 5.5: similar requests ditribution

5.2 Cache replacement and next-server determination

Section 5.1 illustrates how requests are distributed to their primary host. CIC directs the requests to their primary host based on the scoreboard mechanism. When a primary host receives a primary request, it registers the request in the registry. The registry includes all necessary information of primary requests including a movie id, the arrival time of the request, interval, and actual host that provides the request with cache space. (It might be the primary host itself, but if it cannot accommodate the request in its cache, CIC looks for available cache in the cluster.)

The host first looks up the existing entries of the registry to find the preceding request of the new request. Because the host maintains information of all the requests to the same movie, the entry of the preceding stream should be in the registry. The host calculates the interval time between the preceding stream and the current stream, and checks if the interval can be accommodated in its cache space using the interval caching policy. That is, if cache has enough space to accommodate the interval, the interval is cached. If not, the largest cached interval in the cache is compared to the newly created interval. If the largest cached interval (called the victim) is larger than the new interval, the victim is evicted from the cache and the new interval replaces the victim.

If the host fails to find its own cache space for the primary request, it looks for another host that has cache space. The procedure to find another host should be carefully designed so that the amount of probing to find one would not cause a bottleneck and slow system performance. We explore three alternative approaches to find the next host:

1. To find the next host based on the scoreboard of the request. With this approach, the primary host finds the second host listed in the scoreboard and sends the request to the corresponding host. If the second host in the scoreboard fails to find cache space, it forwards the request to the third one listed in the scoreboard, and so on. Figure 5.6 illustrates this approach.
2. To find the next host based on a static list of all the hosts in round-robin fashion. With this approach, every host maintains a static list of all the hosts in the cluster. Once a

host fails to find cache space for the request, it finds the next host in the list in a round-robin manner, and forwards the request to the next host. Figure 5.7 illustrates this approach.

3. To find the next host by randomly selecting a host in the cluster. With this approach, a primary host randomly selects another host, and the randomly chosen host conducts exactly the same procedure to test the possibility to accommodate the request. If it cannot serve the request from the cache either, another host is randomly chosen. Figure 5.8 illustrates this approach.

With each approach, the procedure to find the next host continues until a host is found or it is determined that no host is available for the request. In either case, the decision is sent to the primary host of the request and is reflected in the registry.

We compare the performance of these three approaches in terms of the average number of cached streams and amount of probing to find the next host. The performance analysis shows that the random-selection approach outperforms the other two approaches. The discussion on the performance analysis appears in chapter 7.1

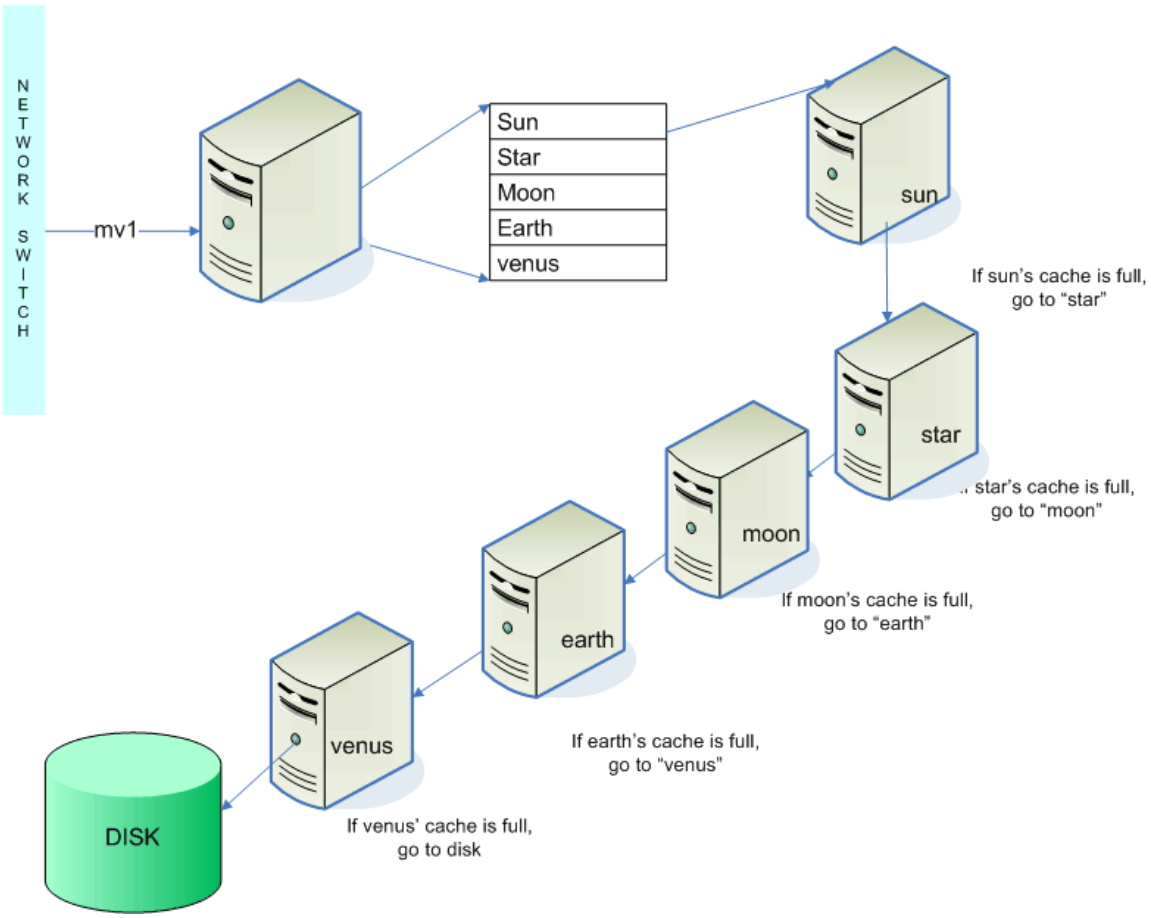


Figure 5.6: scoreboard & scoreboard approach

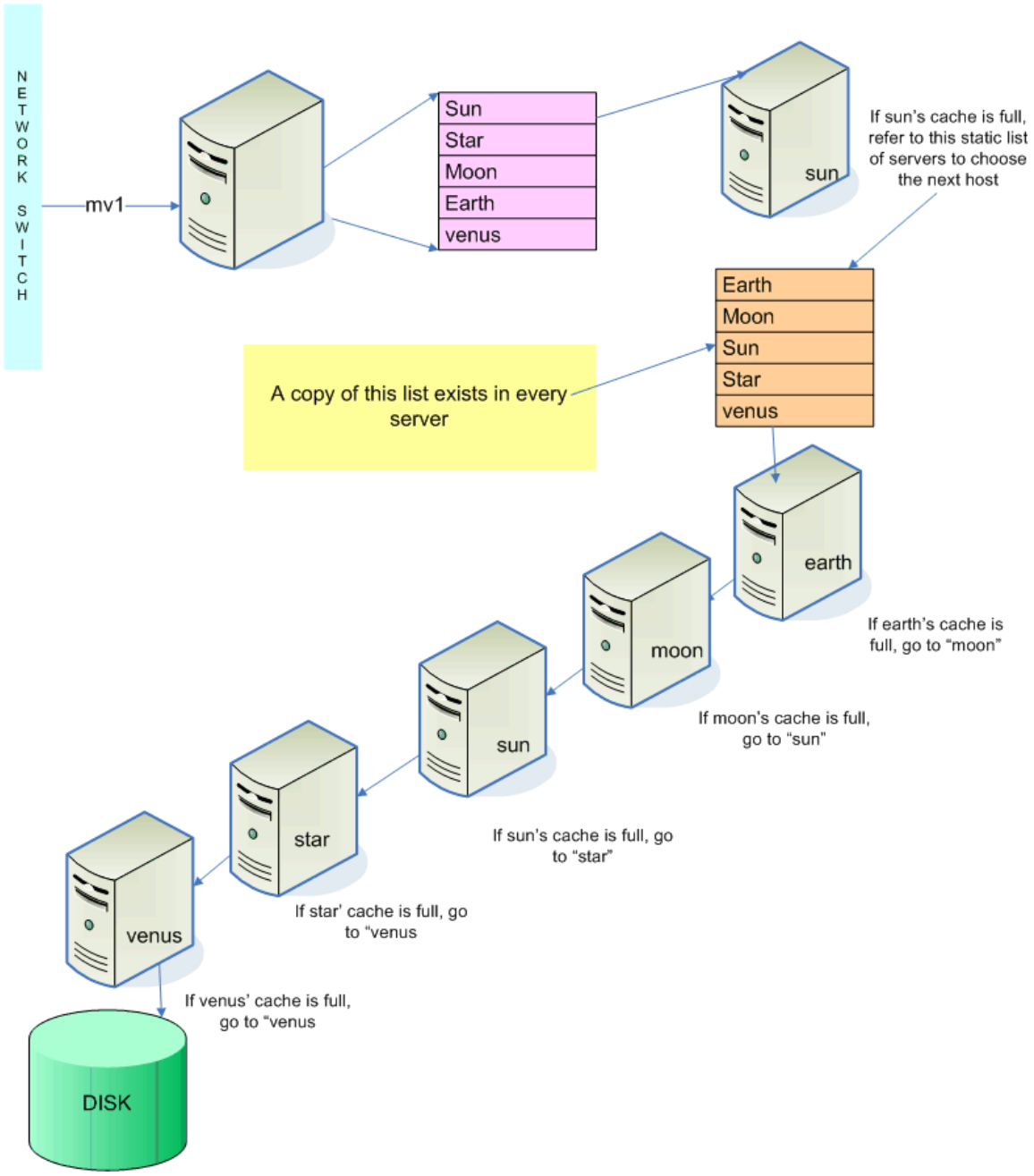


Figure 5.7: scoreboard & round robin approach

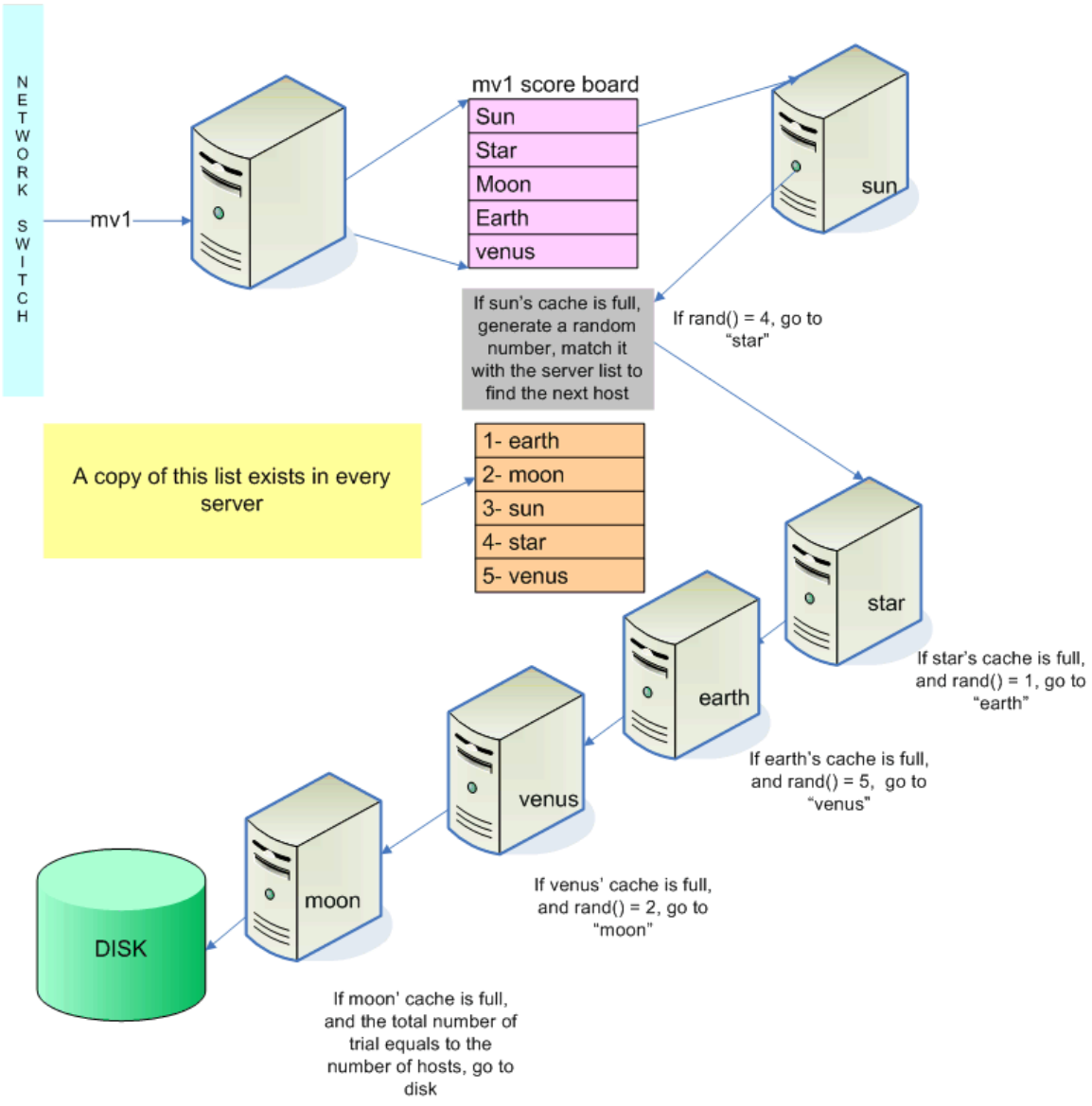


Figure 5.8: scoreboard & random approach

5.3 Scalability

Considering the rapid growth of VOD services, a video server needs to be scalable to service the dramatically increasing number of requests. This section describes how a clustered server can be smoothly scaled using CIC.

Suppose the clustered server currently consists of *n* hosts. Each host is in charge of serving requests to a certain set of movies as primary host.

When a new host is added to this existing clustered environment, the primary host for requests will change. Suppose a new server named *newhost* is added to the cluster. Figure 5.9 shows a simple example of how this addition affects the scoreboards. Among these 5 scoreboards, 4 of them will perform as usual, since the primary hosts are not changed (*mv1*, *mv3*, *mv4*, and *mv5* are directed to *star*, *star*, *sun*, and *earth* respectively as normal). The scoreboard *mv2 sb* has *newhost* as the primary host. That means that any new requests for *mv2* will be directed to a host different from the host to which their preceding streams were forwarded. Since *newhost* does not have any information of preceding requests, all new requests will get data from disk, and we will see a temporary drop in performance. The waiting time to get back the normal performance is considered acceptable because requests come seconds apart, and preceding streams will be quickly established again. We adopt a simple remedy for this problem by letting it be resolved by itself.

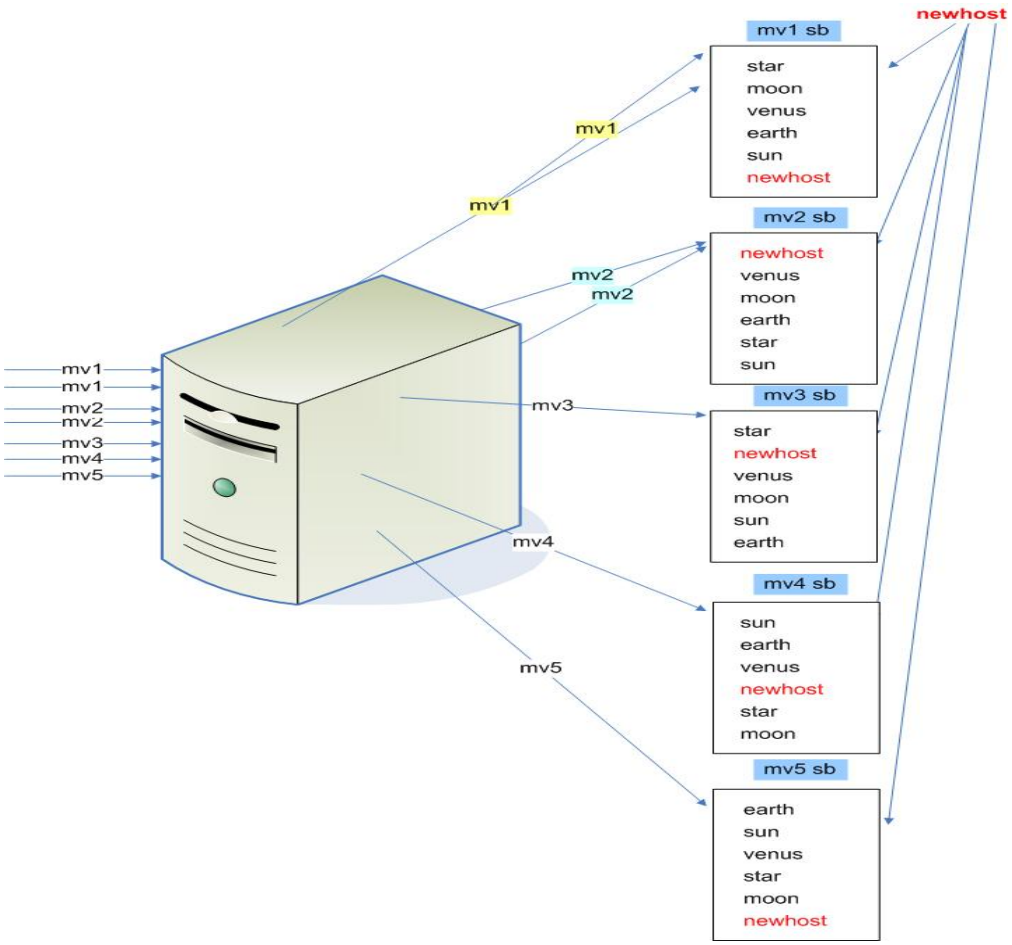


Figure 5.9: adding a new server to the cluster

Similarly, if we plan to remove a server from the cluster, CIC will update the list of servers from all machines. A new scoreboard will be created without the removed server, which means requests that used to go to it will be moved to the next server in the scoreboard. We might see a temporary drop in performance since all new requests sent to the removed server will loose their preceding streams (if they had any). That performance issue is temporary and will be resolved by itself within the movie length interval (worst case).

6 The Simulator

We develop a simulator to conduct a performance study on the proposed cooperative interval caching for a video server cluster. The simulator is written in CSIM 19, a process-oriented discrete-event simulation language for use with C programs. This section includes a brief introduction to CSIM 19 followed by the description of our simulation model.

6.1 Process-oriented discrete-event simulation

A process is an independent program or procedure that is defined by the programmers. If a process generates one event, and completes it before generating another one, it is called an event-oriented simulation. On the other hand, if it generates different threads of events running in parallel, it is called process-oriented simulation.

To reflect the states of CIC system, we use CSIM, a process-oriented discrete-event simulation, in which the state of the system takes discrete values, and changes only at a discrete set of points in time. According to Dr. Raj Jain in his book, *The Art of Computer Systems Analysis*, "A model is called a *continuous*- or *discrete-state* model depending on whether the state variables are continuous or discrete." Dr. Raj Jain provides an example in which "continuous-state models are used in chemical simulations where the state of the system is described by the concentration of a chemical substance". Meanwhile, discrete-event models are used in computer system since the state of the system is "*described by*

the number of jobs at various devices" [32]. Discrete-state models are also known as *discrete-event* models.

We use CSIM to model the simultaneous requests of many VOD events in parallel. In this model, many VOD are requested at a discrete set of points in time. While serving one request, the simulator generates others and serves them simultaneously. In this model, the number of streams being served, the resources used to serve them are variables specifying system state.

6.2 CSIM Objects

In order to help programmers modeling almost every structure, CSIM 19 provides a set of objects. Below are some main CSIM19 objects, described by [33]

- Facilities: model system resources, like disk(s) or cpu(s)
- Storage: model system resources which can be partially allocated to different requests. Memory can be a good example of storage. The difference between facilities and storage is the counter that storage must have to keep track of resource use, and a queue containing processes waiting for resources from the storage.
- Buffers: similar to storage but having two queues, representing two different operations. The first queue contains processes waiting to get tokens from buffer, and the other queue contains processes waiting to return tokens back to the buffer (a token is a buffer unit).
- Events: a process consists of a time ordered set of events. Events are means to synchronize and control interactions between different processes. Even has either one of two states: occurred or not-occurred, which can be changed by the process. In many other cases, a process can be put on hold waiting for a certain event to occur.
- Mailboxes: each process has its own environment. To communicate among processes, CSIM 19 uses mailboxes. Processes deposit messages to, and retrieve messages (data) from mailboxes.

6.3 Simulation Time

SIM time does not relate to CPU time. When the first request comes, a process is generated to handle that request, and simulation time is set to 0. The process takes place

until a *hold(x)* statement is invoked. At that point, the control is transferred to any waiting process, and time will be advanced *x* units. The time unit is based on the nature of the model. For example, if there will be a request every 10 seconds, then we set the SIM time unit to second, conceptually. Not all requests come exactly 10 seconds apart, so SIM function *exponential (interval time)* will generate a fluctuate interval around that 10s. If we want to model a number of processes started simultaneously, value of *x* should be set to 0.

The following example from *http://www.mesquite.com/products/csimprim.htm*, shows how simulation time works:

```
1.  void otherProcess() // the starting process
2.  {
3.      long i;
4.      create("other");
5.      for(i = 0; i < 10; i++) // invoke 10 instances of cust
6.          customer();
7.      hold(25.0);
8.      // other statements
9.  }
10. void customer() // the started process
11. {
12.     create("customer"); // <--- Focus of discussion
13.     // other statements
14. }
```

Line 12 creates a customer upon the call of line 6. After create the customer, the control is back to line 6 until all 10 customers are created. When *otherProcess* executes the *hold(25.0)* statement, it is suspended and the first *customer* process enters the Computing state and continues computing until it suspends itself. At this point, the second *customer*

process starts computing, and so on. It is important to note that in simulation, computing takes no time.

6.4 Confidence Intervals and Run Length Control

Another concept widely used in CSIM is confidence intervals and run-length control. “A confidence interval for a statistic is a range of values in which the true ‘answer’ is believed to lie with a high probability.” [34]. In other words, the confidence interval tells how accurately the data produced by the sample is, compared with the data generated by experimenting on the whole population (if that can be done). For example: we take a sample of 200,000 people to find out how much they spend on internet shopping per year. The mean of the sample is $400. How accurate is this number? How much confidence can we say that this number is 90% close to the number we get if we can possibly survey all people in the world? If we want to achieve the 90% accuracy level, how big should the sample be?

CSIM has answers for these questions. CSIM allows the modelers to set the confidence interval, then it runs the simulation until the goal is met; then it terminates, using its run-length control algorithm. CSIM guarantees that it does not run a simulation for too short an amount of time, which would result in performance statistics that are highly inaccurate. It also guarantees not running a simulation for an unnecessarily long amount of time, which wastes computing resources and delays the completion of the simulation study.

The following example and its interpretation, provided by CSIM19 vendor, explains how run length control works:

#include "csim.h"

1. TABLE tbl;
2. void gen();

```
3.  void sim()
4.  {
5.      create("sim");
6.      tbl = table("tbl");
7.      table_histogram(tbl, 10L, 0.0, 10.0);
8.      table_confidence(tbl);
9.      table_run_length(tbl, 0.01, 0.95, 10.0);
10.     gen();
11.     wait(converged);
12.     report();
13. }
14. void gen()
15. {
16.    create("gen");
17.    while(1) {
           record(expntl(1.0), tbl);
           hold(0.0);
       }
18. }
```

Line #9: the second parameter is the *accuracy* parameter, which specifies the maximum relative error that will be allowed in the mean value of this performance measure. A value of 0.1 is usually used to request one digit of accuracy, 0.01 is used to request two digits of accuracy, and so forth. The third parameter is the *conf_level* parameter, which specifies the confidence level and usually has a value between 0.90 and 0.99. The last parameter is the *max_time* parameter, which places an upper bound on how long the simulation will run. If the specified accuracy cannot be achieved within this time, the simulation will terminate and a warning message will appear in the report.

6.5 Simulation Model

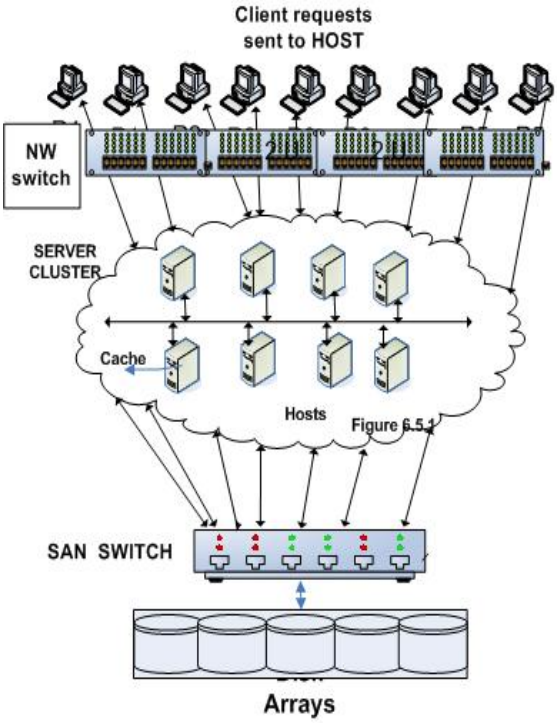


Figure 6.5.1

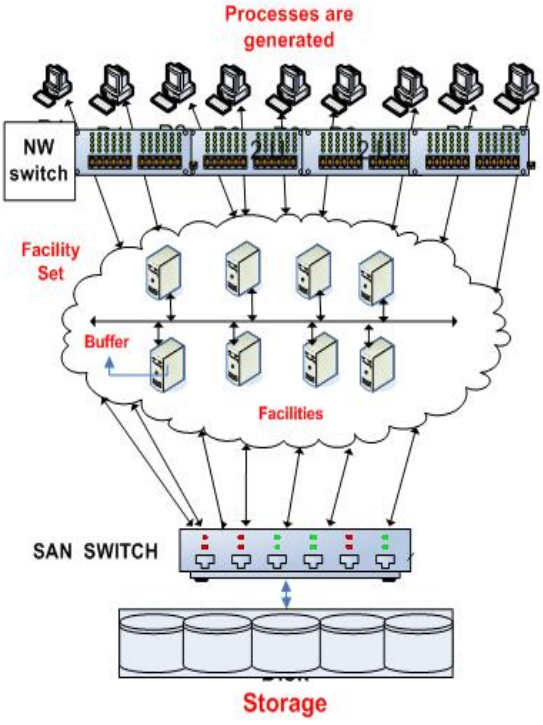


Figure 6.5.2

The above two figures explain how we use CSIM in this project. The left figure illustrates our system model. The right figure illustrates the simulation model, in which:

- Request is represented by ***process***
- Server Cluster is represented by ***facility set***
- Host is represented by ***facility***
- Cache is represented by ***buffer***
- Disk array is represented by ***storage***

Two CSIM objects widely used in our model are facility set and buffer. The following diagram shows how the CSIM objects interact with each other in CIC. At the very top level is the *sim()* process, which is similar to *main()* function.

Figure 6.1 shows how CIC simulation model works:

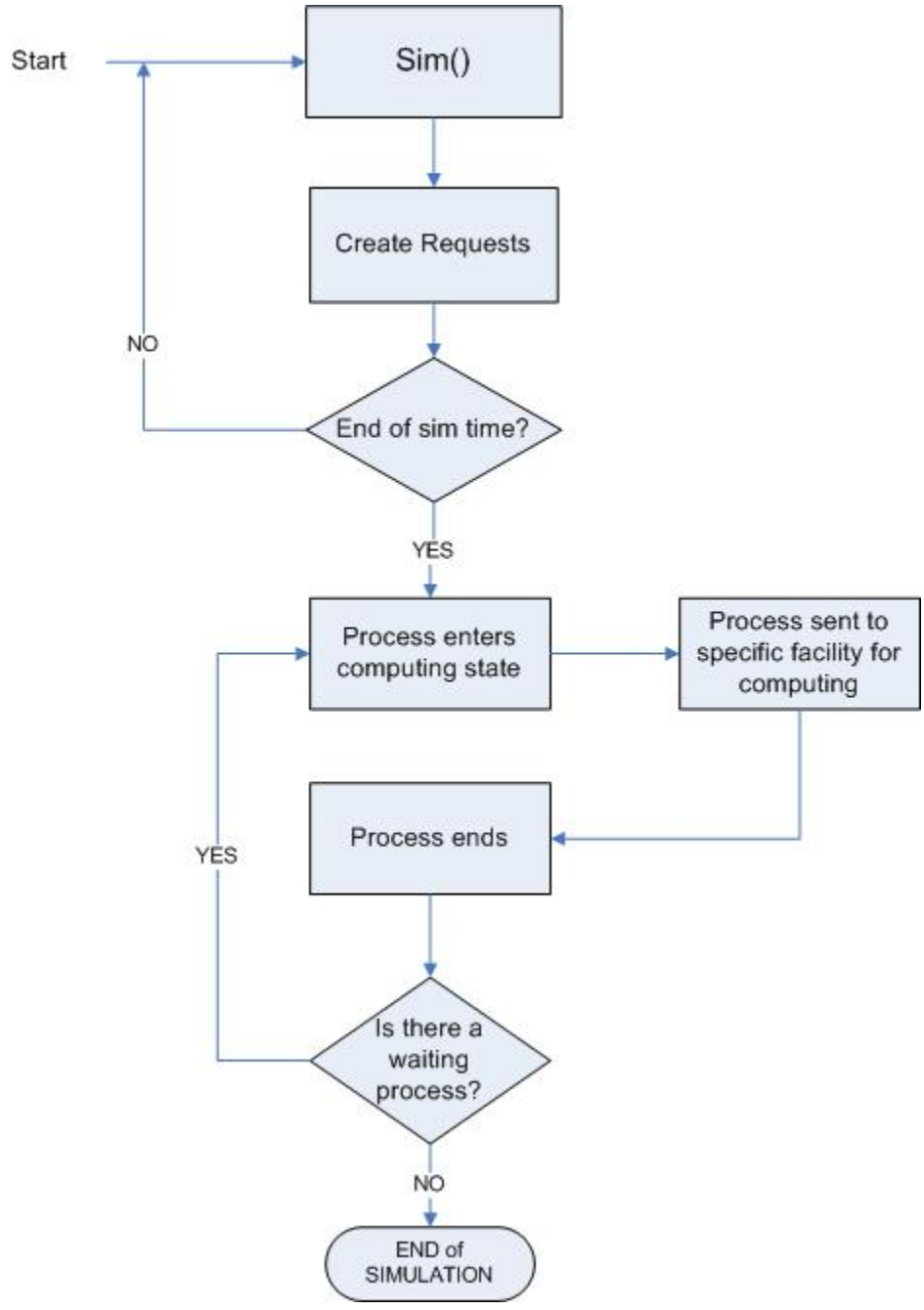


Figure 6.1: simulation model

CSIM does not need a main() function to control program flow. Instead, it uses the sim process, which acts exactly like main(). The sim process is the first process in any CSIM

program. In our implementation, it resides at the host system. It creates a sub process *request* to generate movie requests. Each request is sent to a facility, identified by CIC. Facility communicates with either buffer or storage to get data back to the clients.

7 Performance Evaluation

This section describes the performance analysis of the proposed CIC algorithm. In this performance study, the average number of cached streams is used as a performance metric to indicate the capacity improvement of servers employing CIC. We also measure the number of hosts to probe before finding the final host to serve the request or determining there is no host to serve the request. We can estimate overhead of using CIC based on the number of hosts to probe.

7.1 Comparison of different combinations of request assignment and next server selection approaches

How to assign a request to the primary host and how to select the next server when the current host cannot accommodate the request are two major factors that will affect the performance of CIC. We consider three different alternatives for each factor: using a scoreboard, following a static list of hosts in round robin fashion, and randomly selecting one of the hosts. We will have *SB*, *RR*, and *Rand* denote these approaches, respectively.

Table 7.1 summarizes the parameter values in this performance study. We tried to set the parameters to realistic values. For instance, the number of hosts is chosen to be 8 considering that the 8 port SAN switch is very common in practice. Cache size is chosen to be 1GB per server because most COTS (Commodity Off The Shelf) servers are equipped with about 1GB cache. We assume the movie requests follow a zipf distribution with a skew coefficient 0.27, because this zipf distribution and 0.271 have been used in many research works to describe the distribution of movie requests. In this experiment, the values of all parameters are fixed to find the effect of different combinations of request assignment and next server selection approaches on the server performance.

Parameter	*Value*
#Hosts	8
#Movies	1000
Total Cache Size	8GB
Inter-arrival Time	2 seconds
Movie Duration	90 minutes
Delivery Rate	MPEG1 1.5Mbps
Skewness	Zipf distribution, skew coefficient = 0.27
Request assignment & next server selection approaches	SB&Rand; SB&RR; SB&SB; RR&SB; RR&RR; RR&Rand; Rand&SB; Rand&RR; Rand&Rand

Table 7.1: system and workload parameters

The table 7.1.2 shows the average number of cached streams for different combinations of approaches

Combinations of different approaches	**Number of cached streams**
SB&SB	227
SB&RR	227
SB&Rand	383
RR&SB	234
RR&RR	234
RR&Rand	158
Rand&SB	172
Rand&RR	172
Rand&Rand	166

Table 7.2: Number of cached streams for different combinations of request assignment and next server selection approaches

We also calculate the cache hit ratio which is the average number of cached stream divided by total number of concurrent requests being serviced in the clustered server. With the given set of parameter values, the simulation data read about 2700 for the total number of streams concurrently being serviced in the stream. The number matches with the number of expected streams calculated using Little's Law [37]. Using Little's Law, the mean number of streams NS can be estimated as follows.

NS = arrival rate * movie length = 1/2 * 5400 = 2700 streams

The performance results are presented in the figure 7.1 and the figure 7.2

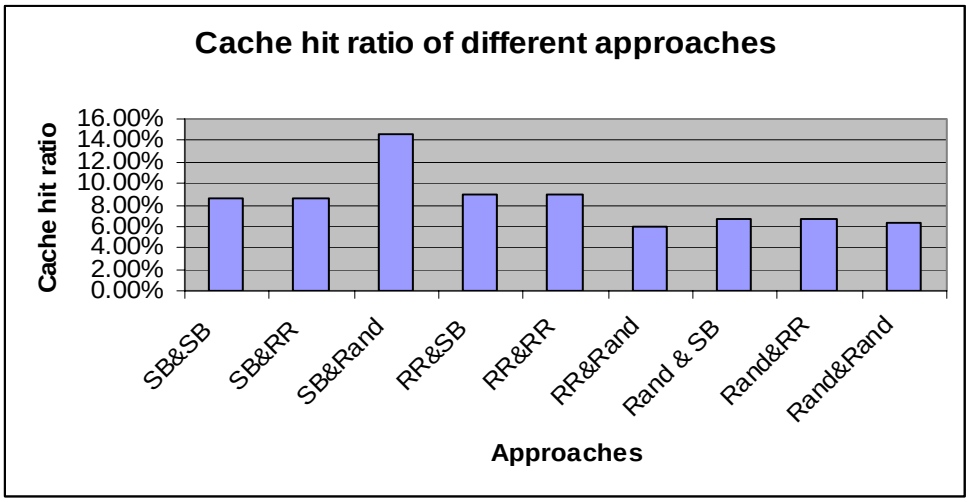


Figure 7.1: cache hit ratio of different combinations of approaches (8GB cache, 8 servers)

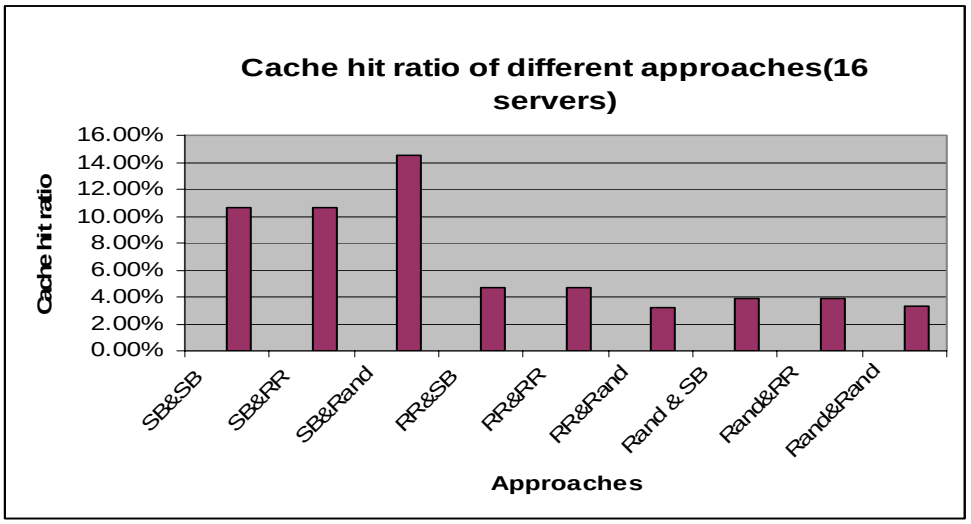


Figure 7.2: cache hit ratio of different combinations of approaches (16GB cache, 16 servers, and the same values for other parameters)

The performance results show that the SB&Rand approach outperforms other approaches with a cache hit ratio of 15%.

The Figure 7.3 shows how many hosts need to be probed until the final host to service the request is found or until it is determined that the request should be serviced from the disk.

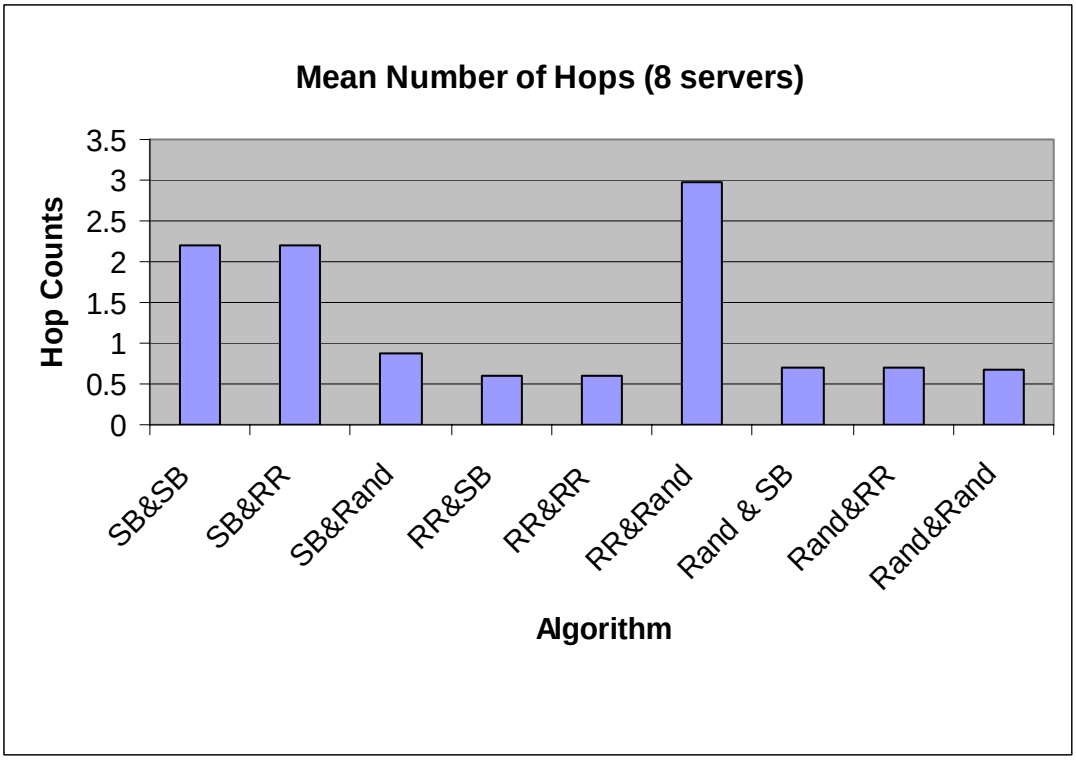


Figure 7.3: Mean Number of Hops

The performance results show that among three alternatives, the SB&Rand approach produces the highest average number of cached streams and RR&SB requires the smallest number of hops to find the final server or to determine that there is no available server. Their performance difference is 68% for the number of hops and 62% for the average number of cached streams (figure 7.4). Probing requires memory access while losing cached streams increases the use of disk bandwidth. Considering disk access is slower than memory access, we believe increasing one memory access is worth to save one disk bandwidth. Therefore, we consider the SB&Rand approach is the best among the three alternatives we experimented in this performance study.

	SB&Rand	RR&SB	Performance difference
Number of hops	0.87519	0.598182	68.60%
Number of cached streams	383	234	61.50%

Figure 7.4: performance differences between RR&SB and SB&Rand

7.2 Impact of cooperation

To see the impact of cache cooperation on the server performance, we conduct another experiment using the parameter values of table 7.1. In the non-cooperative environment, individual servers exploit interval caching, but they don't cooperate. That is, if a server cannot serve a request, the request is immediately sent to the disk. With cooperative caching, if a current server cannot service a request, it exploits CIC to find any available server in the cluster. If no server can accommodate the request in its cache, the request is finally sent to the disk. The performance results indicate that cooperative caching outperforms non-cooperative caching with 95% of more cached streams.

Server id	# of cached streams
Server1	7
Server2	13
Server3	12
Server4	11
Server5	59
Server6	21
Server7	17
Server8	61

Table 7.3: non-cooperative caching

Server id	# of cached streams
Server1	20
Server2	26
Server3	26
Server4	24
Server5	120
Server6	45
Server7	33
Server8	96

Table 7.4: cooperative caching

7.3 Impact of total cache size

To see the impact of total cache size in the cluster, we conduct another experiment using the parameter values of table 7.1; with total cache size varies from 1GB to 8GB. Figure 7.5 shows the result.

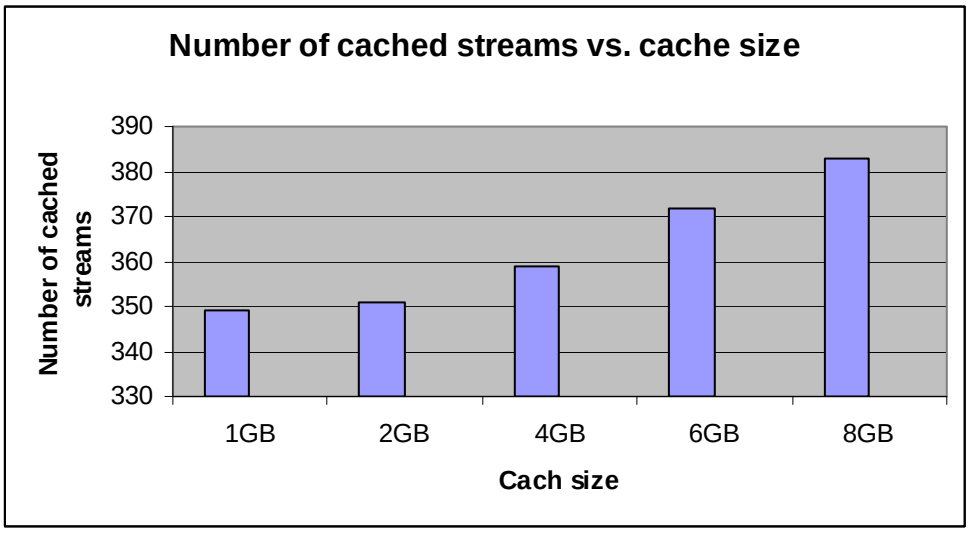


Figure 7.5: number of cached streams vs. cache size

As the total cache size in the cluster increases, the number of cached streams is also increased. One notable observation from Figure 7.5 is that the increment is not linear. As the cache size increases, the number of cached streams is increased, but the possibility of getting larger intervals is also increased. That explains why the increment of cached streams is slower than the increment of cache size.

7.4 Impact of clustering

In this experiment, we use the same parameter values of Table 7.1 while varying number of servers in the cluster. The total 8GB cache space is evenly distributed among *n* servers. The performance results show that the average number of cached streams in the clustered environment is about the same or greater than that of a single server environment with the same total cache size.

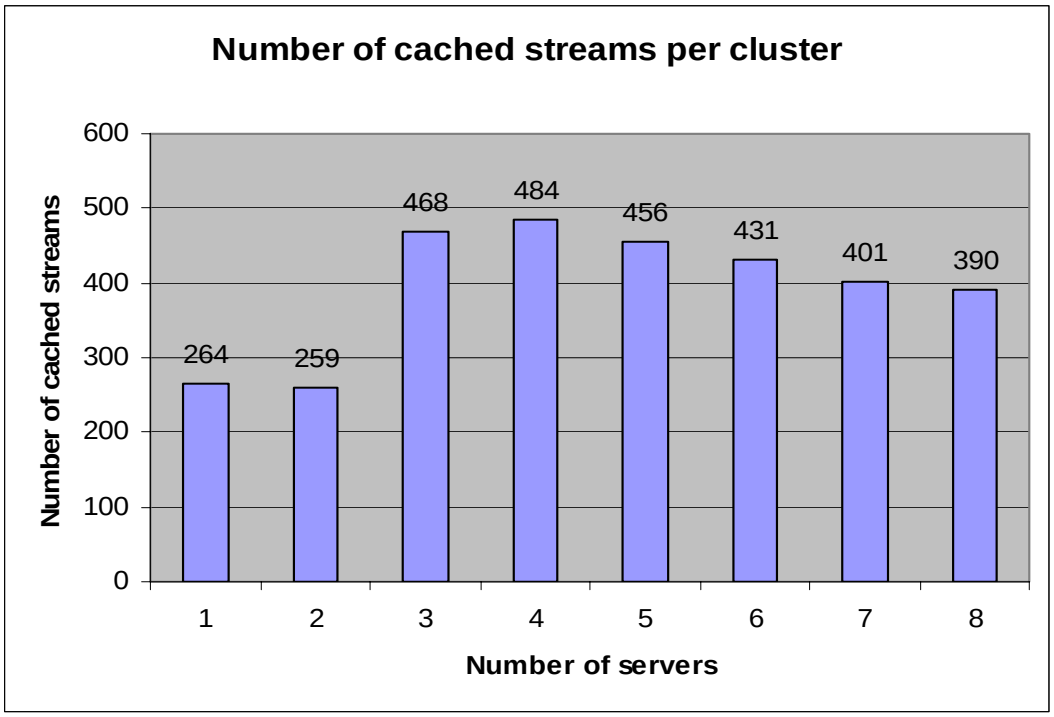


Figure 7.6: Number of cached streams in a cluster environment

We analyze the performance results and analyze the case where the clustered server can accommodate more streams in the cache than a single server with the same cache size as the integrated cache size of the clustered environment. Suppose only 10M of cache space left in the server and the size of largest cached interval is 8M. Let's consider a sequence of newly formed intervals with sizes 6, 4, 4, 2, 2, and 7 attempting to be cached. The table 7.3.4 shows how many streams can be cached out of 6 intervals in a single server environment. In the table, AT, CA, CR, D, SB, and R denote "arrival time", "available cache space", "required cache space by an interval", "stream is served by disk", "stream is served by', and "cached interval that replaced the victim", respectively. In the single server environment, 5 requests could be cached out of 6. After processing all six requests, the largest cached interval size is 6MB.

AT	CA	CR	SB
….			
T1	10	6	C
T2	4	4	C
T3	0	4	R
T4	4	2	C
T5	2	2	C
T6	0	7	D

Table 7.5: cache replacement in an integrated environment

Table 7.6 shows how the same requests can be served in a clustered environment with 3 servers. In the table, MCS denotes “Maximum Cache Size”. Supposed the available cache space of 10 MB is distributed to these three servers with sizes 4, 4, and 2 respectively. A similar sequence of newly formed intervals with sizes 6, 4, 4, 2, 2, and 7 attempting to be cached. In the integrated environment, 5 out of 6 streams are cached, as shown in table 7.5. In this cluster environment, each server has a different maximum cached size stream, which changes the cache replacement values. The result is that all 6 streams are cached, as shown in figure 7.6

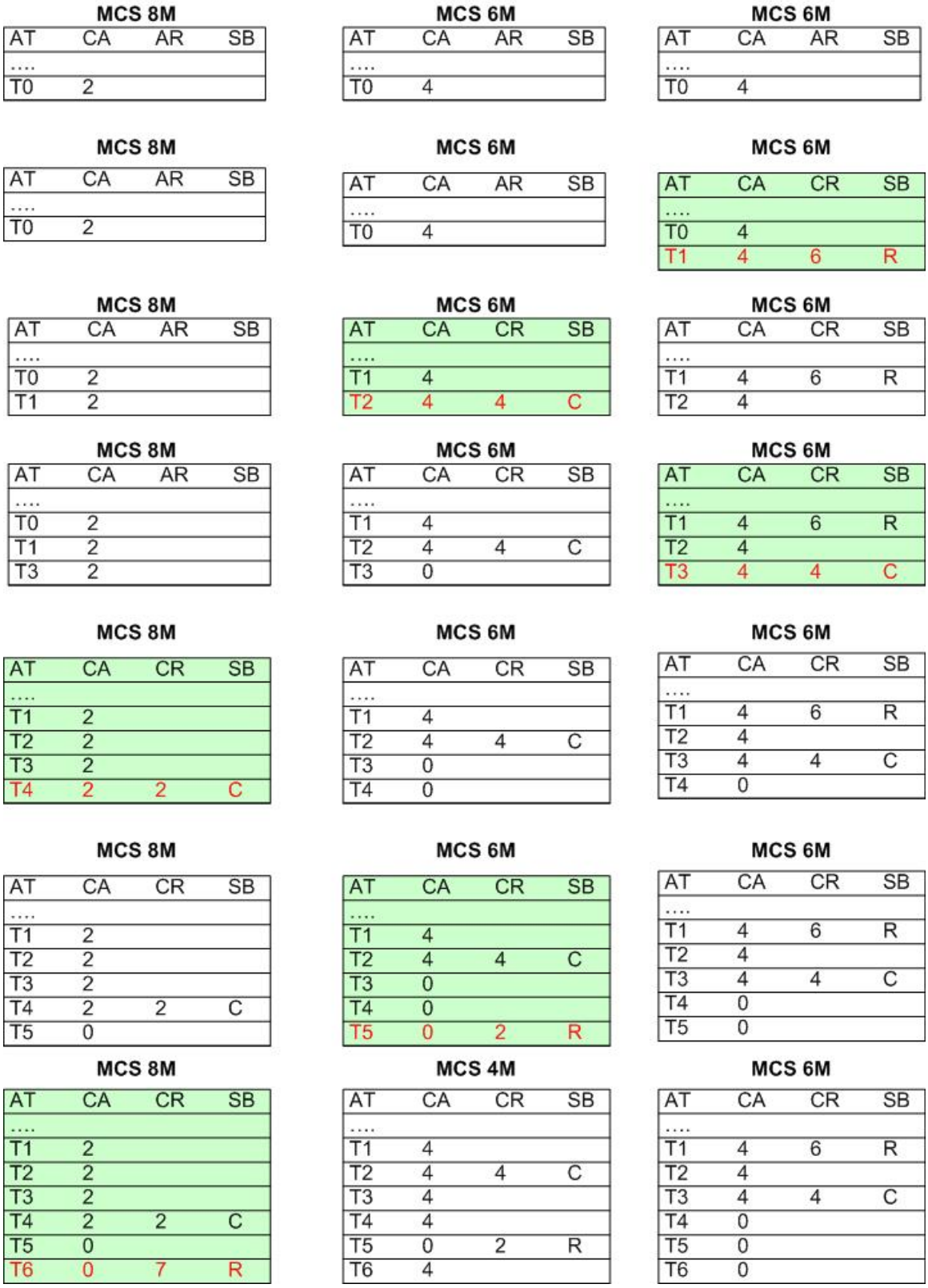


Table 7.6: cache replacement in a distributed environment

8 Conclusion and Future Works

In this project, we design a cooperative interval caching (CIC) algorithm for clustered video servers, and evaluate its performance through simulation. The CIC algorithm describes how distributed caches in the cluster cooperate to serve a given request. With CIC, a clustered server can accommodate twice (95%) more number of cached streams than the clustered server without cache cooperation. There are two major processes of CIC to find available cache space for a given request in the cluster: to find the server containing the information about the preceding request of the given request; and to find another server which may have available cache space if the current server turns out not to have enough cache space. The performance study shows that it is better to direct the requests of the same movie to the same server so that a request can always find the information of its preceding request from the same server. The CIC algorithm uses scoreboard mechanism to achieve this goal. The performance results also show that when the current server fails to find cache space for a given request, randomly selecting a server works well to find the next server which may have available cache space. The combination of scoreboard and random selection to find the preceding request information and the next available server outperforms other combinations of different approaches by 86%. With CIC, the cooperative distributed caches can support as many cached streams as one integrated cache does. In some cases, the cooperative distributed caches accommodate more number of cached streams than one integrated cache would do. The CIC algorithm makes every server in the cluster perform identical tasks to eliminate any single point of failure, there by increasing availability of the server cluster. The CIC algorithm also specifies how to smoothly add or remove a server to or from the cluster to provide the server with scalability.

Possible future works related to this project can be developing analytical model of CIC, extending CIC for hierarchical cache structure where caches at higher level are tried first before forwarding a request to the caches at lower level, and extending CIC for video servers that allow users to generate VCR operations.

9 References

[1] Asia Media, Internet source, http://www.asiamedia.ucla.edu/print.asp?parentid=42032

[2] S. Viswanathan and T. Imielinski. "*Pyramid Broadcasting for video on demand service*". In IEEE Multimedia Computing and Networking Conference, Volume 2417, pp 66-77, San Jose, California, 1995.

[3] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. "*A permutation-based pyramid broadcasting scheme for video-on-demand systems*". In Proc. of the IEEE Int'l conf. on Multimedia Computing and Systems '96, Hiroshima, Japan, June 1996.

[4] K. A. Hua and S. Sheu. "*Skyscraper Broadcasting: a new broadcasting scheme for metropolitan video-on-demand systems*". In SIGCOMM 97, pp. 89-100, Cannes, France, Sept. 1997. ACM.

[5] L. Juhn and L. Tseng. "*Harmonic broadcasting for video-on-demand service*"**.** IEEE Transactions on Broadcasting, pp 268-271, Sept. 1997.

[6] L. Juhn and L. Tseng. "*Fast data broadcasting and receiving scheme for popular video service*". In IEEE Transactions on Broadcasting, pp. 100-105, Mar 1998.

[7] K. c. Almeroth and M. H. Ammar. "*The use of multicast delivery to provide a scalable and interactive video-on-demand service*". IEEE Journal on Selected Areas in Communications, pp. 1110-22, Aug 1996.

[8] C. C. Aggarwal, J. L. Wolf and P. S. Yu. "*On optimal piggyback merging policies for video-on-demand systems*". In Proc. 1996 ACM SIGMETRICS Conf. On Measurement and Modeling of Computer Systems, Philadelphia, PA, May 1996, pp. 200-209.

[9] A. Dan, D. Sitaram and P. Shahabuddin. "*Scheduling policies for an on-demand video server with batching*". In Proc. 6th Int'l. Multimedia Conf.(ACM Multimedia '94), San Francisco, CA, Oct 1994, pp. 15-23.

[10] K. A. Hua, Y. Cai and S. Sheu. "*Patching: a multicast technique for true video-on-demand services"*. In Proc. 6th ACM Int'l. Multimedia Conf. (ACM Multimedia '98), Bristol, U.K., Sept 1998, pp. 191-200.

[11] S. W. Carter and D. D. E. Long. "*Improving video-on-demand server efficiency through stream tapping"*. In Proc. 6th Int'l. Conf. On Computer Communications and Networks (ICCCN'97), Las Vegas, NV, Sept 1997, pp. 200-207.

[12] D. Eager, M. Vernon, J. Zahorjan. "*Minimizing bandwidth requirements for on-demand data delivery*". In Proc. 5th Int'l. Workshop on Multimedia

[13] J.Guo, P. Taylor, M. Zukerman, S. Chan, K. S. Tang, E. W. M. Wong, "*On the efficient use of video-on-demand storage facility*". Proceedings of ICME 2003 IEEE, Baltimore, USA, 2003, pp. 329–332

[14] Meng-Huang Lee, "*Disk system design for periodical video broadcast services*", IEICE Electron. Express, Vol. 1, No. 8, pp.204-210, (2004)

[15] Tat-Seng Chua, Jiandong Li, Beng-Chin Ooi, Kian-Lee Tan, "*Disk striping strategies for large video-on-demand servers*" . February 1997 Proceedings of the fourth ACM international conference on Multimedia

[16] Xiaobo Zhou; Cheng-Zhong Xu. "*Optimal video replication and placement on a cluster of video-on-demand servers*". Parallel Processing, 2002. Proceedings. International Conference on 18-21 Aug. 2002, pp. 547 – 555

[17] Xin Liu; Vuong, S.T., "*Supporting low-cost video-on-demand in heterogeneous peer-to-peer networks*". Multimedia, Seventh IEEE International Symposium on 12-14 Dec. 2005 Page(s):8

[18] S. Acharya and B. Smith, "*MiddleMan: A Video Caching Proxy Server*", in Proc. of NOSSDAV '00, 2000.

[19] Ramesh, S., Rhee, I., Guo, K. "*Multicast with cache (Mcache): an adaptive zero-delay video-on-demand service*". Circuits and Systems for Video Technology, IEEE Transactions on Publication Date: Mar 2001. Volume: 11, Issue: 3, pp. 440-456

[20] Geun Jeong Lee, Chi Kyu Choi, Chang Yeol Choi, Hwang Kyu Choi. "*P2Proxy: Peer-to-Peer Proxy Caching Scheme for VOD Service*". Proceedings of the Sixth International Conference on Computational Intelligence and Multimedia Applications (ICCIMA'05) - Volume 00, pp. 272 – 277, year of publication: 2005

[21] A. Dan and D. Sitaram, "*A Generalized Interval Caching Policy for Mixed Interactive and Long Video Workloads*", In Proceedings of Multimedia Computing and Networking (MMCN), San Jose USA, the International Society for Optical Engineering, pp. 344 – 351, 1996.

[22] Xiaobo Zhou, Cheng-Zhong Xu. *"Optimal video replication and placement on a cluster of video-on-demand servers"*. International Conference on Parallel Processing, 2002. Proceedings, pp. 547- 555

[23] A. Dan and D. Sitaram. "*Buffer management policy for an on-demand video server*". Technical report, IBM Research, Yorktown Heights, NY, 1993

[24] Z. Ge, P. Ji, and P. Shenoy. "*A Demand Adaptive and Locality Aware (DALA) Streaming Media Server Cluster Architecture*". NOSSDAV, May 2002

[25] Te-Chou Su, Shih-Yu Huang, Chen-Lung Chan, and Jia-Shung Wang, "*Optimal chaining scheme for video-on-demand applications on collaborative networks*", IEEE Transactions on Multimedia, Volume 7, Issue 5, Oct. 2005, pp. 972 - 980

[26] E. Cohen and H. Kaplan. "*Exploiting regularities in web traffic patterns for cache replacement*". Proceedings of 31st ACM STOC, 1999

[27] Roger Haskin and Frank Schmuck, "*The Tiger Shark File System*", In Proceedings of IEEE 1996 Spring COMPCON, Santa Clara, CA, Feb., pp. 226-231, 1996

[28] S. Sen, J. Rexford, and D. Towsley. "*Proxy prefix caching for multimedia streams*". In Proc. of IEEE INFOCOM, Mar. 1999.

[29] Ethendranath Bommaiah, Katherine Guo, Markus Hofmann and Sanjoy Paul, "*Design and Implementation of a Caching System for Streaming Media over the Internet*", IEEE Real Time Technology and Applications Symposium, May 2000.

[30] S. Gruber, J. Rexford, and A. Basso. "*Protocol considerations for a prefix caching proxy for multimedia streams*". Computer Network, pp. 657- 668, June 2000.

[31] Kun-Lung Wu, Philip S. Yu, Joel L. Wolf, "*Segment-based proxy caching of multimedia streams*", Proceedings of the 10th international conference on World Wide Web, pp. 36-44, May 01-05, 2001, Hong Kong.

[32] R. Jain, "*The Art of Computer Systems Performance Analysis*", John Wiley & Sons, Inc., 1991

[33] Mesquite Software, CSIM19 Product Description, http://www.mesquite.com/products/documents/CSIM19ProductDescription.pdf

[34] User Guide: C : Confidence Intervals and Run Length Control. Document from http://www.mesquite.com/documentation/guide_c/16confide.htm

[35] Dan, A., Sitaram, D., and Shahabuddin, P., "*Scheduling Policies for an On-Demand Video Server with Batching*", In Proc. ACM Multimedia ’94, October 1994, pp. 391-398

[36] Video Store Magazine, Dec. 13, 1992.

[37] R. Jain, “*The Art of Computer Systems Performance Analysis*”, John Wiley & Sons, Inc., 1991, page 512

[38] D. Wessels, K. Claffy, “*Internet Cache Protocol (ICP), version 2*”, Request for Comments: 2186, September 1997, http://icp .ircache.net/rfc2186.txt

[39] Vinod Valloppillil, Keith W. Ross, “*Cache Array Routing Protocol v1.0*”, Internet Draft, 26 Feb 1998, http:// cp.ircache.net/carp.txt

[40] Kien A. Hua, Simon Sheu, James Z. Wang, “E*arthworm: A Network Memory Management Technique for Large-Scale Distributed Multimedia Applications*”, in *proceedings of IEEE INFOCOM '97*. vol. 3, pp. 990-997, 1997

[41] Sheu, S.; Hua, K.A.; Tavanapong, W.; “*Chaining: a generalized batching technique for video-on-demand systems*”, Multimedia Computing and Systems '97. Proceedings, IEEE International Conference on 3-6 June 1997 pp. 110 – 117

[42] Cherkasova, L.; Karlsson, M., ” *Scalable Web server cluster design with workload-aware request distribution strategy WARD*”, Advanced Issues of E-Commerce and Web-Based Information Systems, WECWIS 2001, Third International Workshop on, pp. 212 - 221