San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

2008

Providing VCR Functionality in VOD Servers

Ngyat Tsui
San Jose State University

Follow this and additional works at: http://scholarworks.sjsu.edu/etd_projects

Recommended Citation
Tsui, Ngyat, "Providing VCR Functionality in VOD Servers" (2008). *Master's Projects*. 152.
http://scholarworks.sjsu.edu/etd_projects/152

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

PROVIDING VCR FUNCTIONALITY IN VOD 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

Ngyat Tsui

December 2008

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

____

Dr. Suneuy Kim

____

Dr. Robert Chun

____

Dr. Mark Stamp

APPROVED FOR THE UNIVERSITY

___

Dr. Suneuy Kim

___

Dr. Robert Chun

___

Dr. Mark Stamp

TABLE OF CONTENTS

1 INTRODUCTION....... 5

2 ERMT and BEP 11

2.1 Earliest Reachable Merge Target 12

2.2 Best Effort Patching....... 13

2.3 INTERVAL CACHING IN CLUSTERED MULTIMEDIA SERVERS 14

3 CLIENT INTERACTIVITIES....... 27

4 PROPOSED ALGORITHM....... 33

4.1 Client Local Buffer Utilization....... 33

4.2 Periodical Multicasting....... 34

4.3 Algorithm in Details....... 36

4.4 Implementation – Hierarchical-Stream Merging and Stream Multicasting with Fixed Interval....... 48

5 SIMULATIONS AND ANALYSIS....... 55

5.1 Workload Parameters and Performance Metric....... 55

5.2 Simulations and Results....... 57

6 CONCLUSION AND FUTURE WORKS....... 80

REFERENCES....... 81

APPENDIX A: SIMULATION RESULTS....... 85

APPENDIX B: SOURCE CODE....... 90

1. Introduction

Video-on-Demand (VOD) is one of the fastest-growing Internet services today. Yet it is still considered a challenge to design video servers that can keep up with the steeply increasing needs of VOD services, mainly due to the characteristics of video objects. A video object requires continuous playback at a specified rate (e.g., 4 Mbps for an MPEG2 movie) and a large amount of storage (e.g., 1 GB to store a one-hour-long MPEG1 movie).

Scientists and researchers have proposed numerous algorithms and architecture prototypes to advance video server capacity during the last decade. One of the most prominent approaches is the use of resource-sharing techniques. The idea behind resource-sharing techniques is to have clients share VOD servers' limited hardware resources, such as, cache, disk bandwidth, and network bandwidth, to increase effective server capacity. Representative resource-sharing techniques include batching [3], piggybacking [4], patching [5], resource-based caching (RBC) [6, 7], interval caching [8], and hierarchical-stream merging [9, 10, 11, 12].

Batching [3] is one of the first used resource-sharing technologies. It groups user requests into batches before serving them. Hence, a resource that was once able to support only a single request may now serve a batch of requests. Users in the same batch share not only a video stream but also the same play point. In real world situations, user requests do not arrive simultaneously. In order to form a batch, earlier requests have to be put on hold. Some users may cancel their requests due to the long wait. This process also determines that batching is not capable of true VOD support. On the other hand, implementation of batch technology is very simple and requires minimum calculation on

the VOD-server side. Batching is usually used with other resource-sharing technologies, such as patching.

Since audiences are not aware of the difference when playback rates vary within certain ranges, piggybacking [4] technology slows down a leading stream and speeds up the following stream. Eventually these two streams merge into one. It may take a relatively long time to merge because the applicable playback range is limited. Managing many different playback rates also puts a huge burden on media servers. Therefore, piggybacking technology is not widely applied on VOD servers, compared with other resource-sharing technologies.

Patching [5] is also known as stream tapping. Patching technology tries to exploit a client's local buffer and uses a client's network bandwidth. A client is required to receive two or more video streams from a VOD server at the same time. Current playback uses a patching stream. Data received from target streams are saved in the client's local buffer to support later playback. Hence, patching technologies require the client's local buffer, which should be capable of containing video content as big as the displacement between a patching and a target stream's play point. The patching stream is terminated when clients' playback can be supported by content saved on local buffers from the target stream. This termination of the patching stream is known as a "merge," in technical terms. After the merge, the client refers to only the target stream for the rest of the playback.

The purpose of the interval-caching [8] technique is to accommodate more clients than the disk bandwidth can support. With interval caching, when the interval between a preceding stream and its following stream accessing the same video is cached, data used by the preceding stream are recycled and stored in the cache to feed the following stream.

Therefore, the following stream can receive data from the cache instead of the disk system. Interval caching is able to increase throughput by approximately 20% over a broad range of provided system and workload parameters, in which users generate only normal playback operations. Chaining [20, 21, 22] is a special type of interval caching because instead of caching intervals on the media server, chaining saves the intervals on the clients' side. These clients form "chains" for the same movies. Chaining saves not only cache space but also network bandwidth on media servers. However, chaining is more difficult to manage due to fragments of the media file spreading over many clients.

Hierarchical stream merging [9, 10, 11, 12] is similar to patching in terms of client buffer and bandwidth utilization. A client is also required to refer to two or more video streams from the server at the same time. But hierarchical-stream-merging technology allows every stream to become both a patching stream and a target stream. After a merge occurs, a client will not only refer to a previous target stream but also starts referring to that target stream's target stream until the next merge. Thus, the merging streams can form trees with an unlimited number of layers, while patching technology is capable of building only one- or two-layer stream trees. For a relatively balanced stream-merging tree, the more layers the tree has, the more efficiently resources are being shared. Theoretically, with hierarchical-stream-merging technology, a VOD server's system requirements increase logarithmically with increased user demands. Bandwidth skimming [23, 24] technologies are a subset of hierarchical-stream-merging technology. Common-stream merging requires twice the bandwidth of normal playback, while bandwidth skimming is designed for users whose bandwidth is higher than the playback rate and lower than twice the playback rate.

These resource-sharing techniques rely on the consistent and expected playback behavior of clients as they share resources. However, user interactivity using VCR operations is a highly desirable feature of true VOD services, and it needs to be considered in the design of video servers. Typical VCR operations include rewind, fast-forward, pause, jump-forward, and jump-backward. Though resource-sharing techniques are innovative methods that significantly improve server performance, they suffer from performance degradation when users behave unexpectedly by generating VCR operations. Therefore, researchers have been intensely studying the impact of VCR operations on the performance of resource-sharing techniques and working to develop an efficient way of supporting VCR functionality along with these techniques. Considering the capability of resource sharing to improve VOD-server capacity, the purpose of this project is to research the impact of VCR operations on the performance of resource sharing and to propose an efficient way to support VCR functionality in VOD servers with resource sharing.

Researchers have investigated the impact of VCR operations on the performance of batching [14, 15, 16, 17, 18, 28], piggybacking [14, 17, 18, 28], patching [13, 19, 26, 28], chaining [25, 28, 29], and hierarchical-stream merging [24, 26, 27]. Some researchers offer solutions to support VCR functionality in VOD servers with resource-sharing technology. These solutions can be categorized into the following groups.

- Contingency Pool: References [14, 15, 16, 17, 18, 19] take the contingency-pool approach, which can be used with any resource-sharing technology. The idea is to reserve a pool of channels to support potential VCR operations. The key of this approach is finding a balance between the channel reservations of normal playback

and VCR support. Choosing the right size of the contingency pool becomes extremely important. The dilemma is that a small pool size might not be enough to support VCR operations, while a big pool size might be a waste of resources. Although some researchers claim they have discovered an “optimal balance,” so far no such balance is well accepted by others.

- Client Buffer: Client-buffer utilization is not new to resource-sharing technology. Sources [13, 24, 25, 26, 27, 28] all, more or less, use client buffers. In [29], one can see an extreme case that requires a client local-buffer size as large as a video file, which is equal to a few gigabytes. This size buffer is not very practical, even with more affordable computer hardware, especially because media files are getting bigger and technology is becoming higher definition. Client buffers cannot be the sole method for VCR-operation support. Yet cooperation among clients can minimize the resource requirements toward a central media server. Like the contingency pool, client-buffer utilization can be used with any resource-sharing technology, as well.
- Periodical Multicasting: One intelligent idea [26] is to broadcast the media file periodically with a fixed or flexible interval. With periodical-multicasting technology, the VOD server launches multicasting video streams at scheduled times from the beginning of the media file. These multicasting streams terminate only when the end of the media file is reached. With the right interval size, most VCR requests can be fulfilled by patching into the closest multicasting channel. Although periodical multicasting requires patching or stream merging for VCR support, it can be used with any other resource-sharing technology. Unfortunately, experts have conducted little research into this method thus far.

- Hybrid: The combination of different resource-sharing and VCR-support technologies is a very promising approach. Best-effort patching (BEP) [13] is a good example. BEP combines the client buffer, periodical multicasting, patching, and dynamic merging technology, and has been able to achieve much better performance than the regular protocols that use only one VCR-support technology

Based on thorough studies of existing resource-sharing and VCR support techniques, this researcher proposes an algorithm that can significantly reduce the performance degradation of resource sharing caused by VCR operations. The goal is to enhance a hierarchical stream merging technique, Earliest Reachable Merge Target (ERMT) with VCR support by utilizing client local buffer and periodical multicasting techniques. The performance of the proposed algorithms is evaluated and analyzed through simulations. The performances will be evaluated and compared with the performance of a very competitive published VCR support algorithm, best effort patching (BEP) [13]. All simulators will be implemented with CSIM19 in C language.

The remainder of this report is organized in following way: section 2 introduces the chosen resource-sharing techniques, ERMT and BEP, in detail; section 3 explains client interactivities and chosen VCR support techniques for the proposed algorithms, which are described in section 4. In addition, section 4 shows briefly how these algorithms are implemented. Simulation results and workload parameters are presented in section 5, along with a thorough analysis. Section 6 summarizes the report and discusses possible future studies.

2. ERMT and BEP

This project focuses on one hierarchical stream merging technique, Earliest Reachable Merge Target (ERMT) for following reasons:

1. Unlike some resource-sharing techniques such as batching, hierarchical stream merging is able to satisfy playback requests from clients with zero delay.
2. Compared to other techniques, hierarchical stream merging is able to achieve logarithmic performance due to its tree structures with unlimited layers, which means a client is subject to unlimited number of merges. Piggybacking, regular patching, interval-caching and chaining are capable of sharing system resources only in the linear fashion. Although some patching techniques also form merging trees, they are unable to reach logarithmic performance because of the limited depth of the trees. Clients on the servers using patching techniques merge only once or twice depending on the depth of the merging tree.
3. ERMT has the most outstanding performance [10, 31, 32] among hierarchical stream merging techniques.
4. Few previous studies addressed how ERMT supports client interactivities and how these interactivities affect ERMT's performance. For performance comparison purposes, this researcher chooses Best-Effort Patching (BEP) [13], which was a patching technique specifically proposed to support VCR interactions and admissions without delay. The authors of paper [13] claim

that BEP is able to achieve significantly better performance than conventional VOD protocols that use a dedicated stream for every client request.

The rest of this section describes ERMT and BEP in details.

2.1 Earliest Reachable Merge Target (ERMT) Technique

For a given set of client requests, hierarchical-stream merging technology requires the use of a merge tree. The cost of the merge tree is the total amount of the media data that the server has to send to the set of clients. The optimal merge tree, which has minimum cost, can be achieved using some dynamic programs [31] when each new request arrives. These dynamic programs require much computing time and space. In addition, each client request's arrival time must be known in advance. A more practical approach is to find some near-optimal merge trees with less calculation. Earliest reachable merge target (ERMT) [31] technique builds a merge tree with the closest cost [10, 31, 32] to the optimal merge tree. ERMT is one of early merging [31] technologies that are much simpler when compared to finding the optimal merge tree. The principle of early merging is to merge adjacent streams as early as possible, if they can be merged. The earliest reachable merge target for a given stream is the closest early stream that will last long enough in which the given stream can merge. As illustrated in Figure 1, streams A, B, C, and D started at times 1, 13, 16, and 18, respectively, to fulfill four client requests. Obviously, stream A is stream B's earliest reachable merge target. Stream C chooses stream B as the earliest reachable merge target, because stream B will not merge into stream A until time 25, while stream C will be able to merge with stream B at time 19. Although it is also a reachable merge target for stream C, stream A is not the earliest

reachable merge target. Stream D also chooses stream B as the earliest reachable merge target. It will merge into stream B at time 23. Stream C is not a reachable merge target for stream D, because stream C will merge into stream B at time 19, yet stream D needs stream C to be available until time 20 in order to merge. In fact, the ERMT algorithm extends the ending time of stream B when stream C and stream D choose it as the target. This is not shown in Figure 1 for simplification purposes, as it does not affect the target-choosing process, described above. In addition, lines representing streams B, C and D in Figure 1 should have the same angle as that of stream A because all streams have same transmission speed, which is regular play speed. The rationale of different slopes of streams A, B, C and D here is to show the merge between the streams and their merge target.

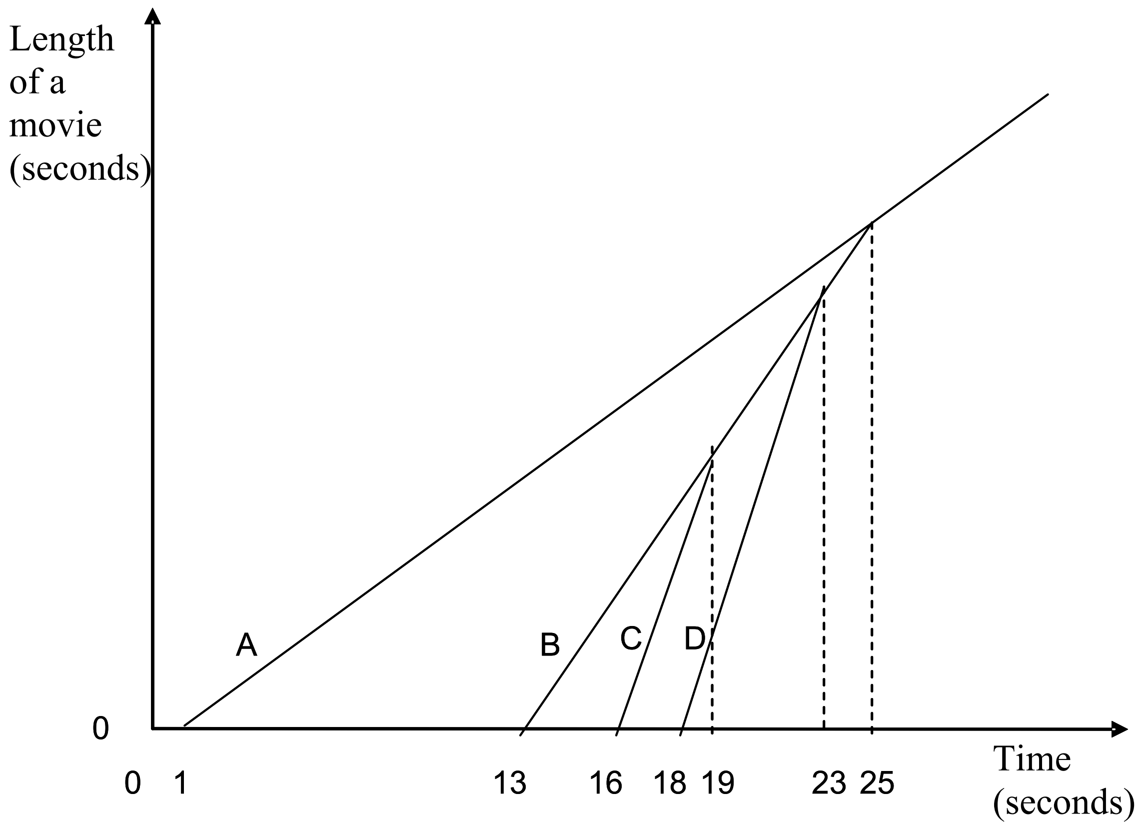

Figure 1: Earliest Reachable Merge Target (ERMT) Example 1

ERMT assumes no future client arrivals while choosing merge targets. Hence, ERMT computes not only a merge target for the new stream but also the chosen target's merge target in order to maintain constant efficiency of the merge tree whenever a new client arrives. The same computation applies to the merge target's newly chosen merge target, which may or may not be the original merge target. As VCR operations are not allowed, for each selected merge target, its lifetime is extended unless the lifetime is no longer extendable. This progression continues by traversing towards the root of the merge tree until no merge target can be found for the stream being computed. In the previous example, stream D is started when a new client arrives; stream B was chosen as merge target. ERMT then computes a merge target for stream B, whose existing merge target is

stream A. Since there is no other stream for B to catch, stream A remains the merge target for stream B. The same computation applies to stream A and ends because no merge target is found. This practice is better illustrated with another example, shown in Figure 2.

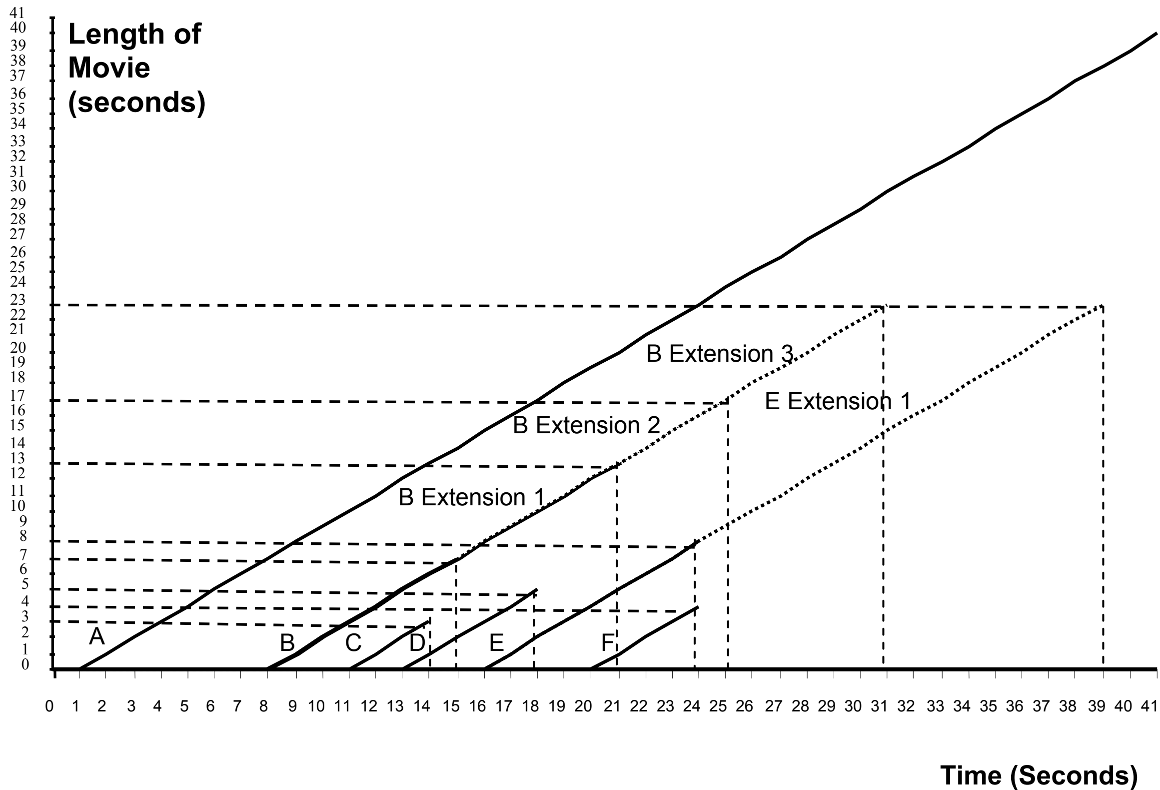

Figure 2: Earliest Reachable Merge Target (ERMT) Example 2

As shown in Figure 2, stream A is the first stream triggered by a client request. Since there are no earlier streams available, stream A's lifetime is set to the same length as the movie's, which is 90 minutes or 5400 seconds. As in Figure 1, the x-axis of Figure 2 represents the timeline, while the y-axis presenting play-points of the movie. During every second, a stream provides a one-second length of movie content at "Play" speed. Thus "Play" speed is always set as 1. Stream B is launched at the 8th second. Stream A is

the only stream whose play-point is earlier than stream B's. It is apparent that ERMT chooses stream A as the merge target for stream B at the moment. ERMT then tries to find a merge target for stream A. The process stops since there is no earlier stream in which stream A can merge. Starting at the 8th second, clients of stream B receive movie content not only from stream B but also from stream A. The movie content from stream B is used for instant display purpose; the content from stream A is saved to a client's local buffer for later display. At the end of the 14th second, when stream B ends, clients of stream B will have watched the first 7 seconds of the movie and saved the 8th to 14th seconds of the movie to the local buffer. From the 15th second, these clients will receive the 15th second's and later frames of the movie only from stream A while watching the content saved earlier in the local buffer. This is the assumption made at the 8th second: that no future client will arrive.

At the 11th second, stream C is initiated. The earliest merge target for stream C is stream B. The gap between these two streams' play-points is 3 seconds, which means it will take 3 seconds for stream C to merge to stream B at the end of the 14th second. Stream B is reachable because it will be available until the end of the 15th second. As the newly chosen merge target, stream B then must find a merge target that meets the following criteria: the merge target should be the earliest reachable into which the latest stream targeting stream B will merge. The latest (and the only) stream targeting stream B is stream C. Due to the fact that each client refers up to 2 streams, stream C's clients can start receiving from the merge target of stream B only after stream C merges into stream B at the end of the 14th second. Stream A is reachable in this case—the gap between its play-point and stream B's is 7 seconds; obviously, stream A will still be available at the

21st second. Hence stream B's merge target, stream A, remains the same. As a result, stream B is extended the first time to the end of the 20th second. Clients of stream C listen to both stream B and A from the 15th to the end of the 20th second and only to stream A afterwards.

Stream D starts at the 13th second. Just like stream C, it chooses stream B as the merge target. Stream B is extended a second time to the end of the 23rd second after choosing stream A again as the merge target because, as the latest stream targeting stream B, stream D will merge into it at the end of the 17th second.

Stream E is created at the 16th second. Similar to the case of stream C and stream D, it selects stream B as the merge target and causes stream B to be extended a third time to the end of the 30th second.

Finally, stream F arrives at the 20th second, and its earliest reachable merge target is stream E, which is supposed to end at the same time as stream F starts its merge. After being chosen as the merge target for stream F, stream E must find a merge target, as well. Its existing merge target is stream B, which will end at the end of the 30th second. The displacement between stream B's and stream E's play-points is 8 seconds, which makes stream B unreachable. If stream E chooses stream B as merge target, then stream F's clients are able to start receiving from stream B only after stream F merges into stream E at the end of the 23rd second. It will take the clients another 8 seconds to receive from both streams E and B before they can solely listen to stream B at the 32nd second. However, stream B ends at the end of the 30th second. Therefore, the existing merge target of stream E, stream B, should not be picked as the merge target of stream E.

With a lifetime same as the length of the movie, stream A becomes the earliest reachable stream for stream E. As a result, stream E is extended for the first time to the end of the 38th second, which is equal to the time stream F merges into it, at the end of the 23rd second, plus its play-point difference from stream A, 15 seconds. ERMT then tries to find a merge target for stream A. The process stops as there are no earlier streams available. Regarding stream E's previous merge target, stream B, its lifetime will not be shortened, although clients of stream E stop referring to it. Figure 2 illustrates the process of how merge trees are built by ERMT and how a stream may change its merge target during the process.

As shown in the example, a stream's lifetime is always extended after being chosen as a merge target. This is due to the fact that no VCR operations are being considered by ERMT. Without VCR operations allowed, each newly arrived client always starts from the beginning of a movie. Thus the play-point of the corresponding stream triggered by the arrival of a new stream is always behind the play-point of any existing stream. Later in this paper, this researcher will show that the lifetime of a chosen merge target may remain the same if ERMT with VCR support is applied.

2.2 Best-Effort Patching (BEP)

Best-Effort Patching (BEP) is an innovative algorithm to support VCR operations on VOD servers. It applies patching techniques between periodically launched multicasting channels. The multicasting technique serves a dual purpose in BEP. First, multicasting is always an essential element of the patching technique. Multicasting channels make the ultimate merge targets, as their lifetime is set to the length of the movie. Second, once

these many multicasting streams are evenly spread along the timeline, VCR operations can be supported more easily. No matter where the play-point is set by any VCR operation, there is always a multicasting stream close by that can serve as a merge target. The unique patching algorithm used by BEP forms stream-merging trees with two layers, as shown in Figure 3. The first layer is made of patching streams directly targeting periodical-multicasting streams. The second layer is composed of patching streams aimed at the first-layer streams. Experts who developed BEP proposed an algorithm called dynamic merging [13], which can help newly started streams choose a merge target between a multicasting stream and a stream from the first layer of an existing merge tree, depending on potential costs, when a new request takes place. Dynamic merging also decides whether to extend the lifetime of a first-layer stream after it’s chosen as a merge target.

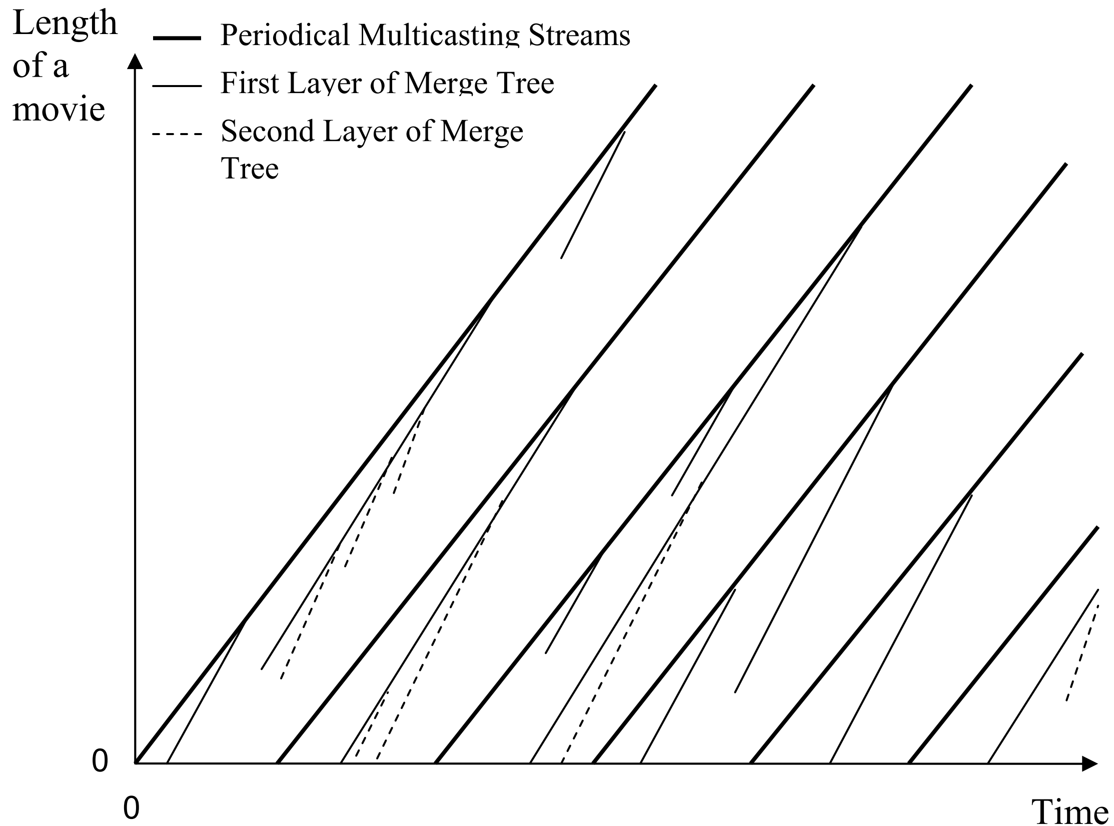

Figure 3: Best-effort Patching (BEP)

For each newly launched stream, BEP first finds the periodical multicasting stream with the closest play-point among all periodical multicasting streams whose play-points are earlier than the new stream. If there are no other streams utilizing the closest periodical multicasting stream as the merge target, then the new stream will choose this multicasting stream as merge target. Otherwise BEP searches through all the first layer streams of existing merge trees, targeting the periodical multicasting stream. If none of these streams are reachable, the closest periodical multicasting stream is selected as the new stream's merge target. The definition of "reachable" here is same as the definition in ERMT—the candidate's lifetime is long enough into which the new stream can merge. An insignificant difference is that if a candidate's lifetime ends at same time the merge

from the new stream takes place, the candidate stream is considered reachable in ERMT while this marginal case is judged as not reachable in BEP. This is described as the first part of dynamic merging technique.

The second part of dynamic merging is applied when there are multiple reachable first layer streams of existing merge trees targeting the periodical multicasting stream. For each of these streams, BEP chooses the smaller number of the play-point displacement from the candidate stream to its targeting periodical multicasting stream or the lifetime left of the candidate stream after new stream merges. This number represents the exact amount of stream time saved if the candidate stream is chosen as the new stream's merge target rather than choosing the periodical multicasting stream as the new stream's merge target.

The reason is quite simple. If the new stream selects the periodical multicasting stream as the merge target instead of any first layer stream, then the cost of merge is the play-point displacement between the new stream and the periodical multicasting stream. This stream time cost is called c1. On the other hand, two possibilities exist if a reachable first layer stream is picked as the merge target of the new stream. The first possibility is the merge target does not need to be extended. In this case, the merge cost of the new stream is only the play-point displacement between the new stream and the candidate stream. This stream time cost is called c2. The second possibility is that a lifetime extension of the candidate stream is necessary if it is chosen as a merge target. The corresponding merge cost is the extended candidate stream time plus c2. This sum of stream time costs is called c3. Therefore, the cost saved by choosing this candidate

stream, rather than the periodical multicasting stream, is either c1 minus c2 or c1 minus c3. These are named saving s1 and saving s2, respectively.

S1 is the play-point displacement from this candidate stream to the periodical multicasting stream, whereas s2 is the lifetime left of this candidate stream after the new stream merges in and before the lifetime extension. If s1 is less than or equal to s2, the lifetime extension of the candidate stream is not required. Thus s1 represents the real saving. If s2 is less than s1, the lifetime of the candidate stream has to be extended in order to allow the clients of the new stream to merge into the periodical multicasting stream. In this case s2 represents the real saving. Therefore the smaller number of s1 and s2 always shows the true saving. BEP selects the first layer stream with the biggest saving as the merge target for the new stream.

The third part of dynamic merging is utilized to decide whether a chosen first layer stream needs to be extended in order to support the merge of the new stream. Given that BEP takes only the reachable first layer stream as the merge target, similar to ERMT, the first layer stream's lifetime must be extended at the moment it is chosen as the merge target. However, this is true only while there are no VCR operations allowed. One main function of BEP is to support VCR. With VCR operations, a new request from a client may arrive at any frame of the movie. The play-point of the matching stream triggered by this arrival may, therefore, be earlier than the play-points of some existing streams targeting same first layer stream in a merge tree. In this case, the lifetime of the first layer stream does not need to be extended after being chosen as the merge target of the new stream prompted by the VCR request. The exercise of dynamic merging can be better illustrated with a more detailed example, as shown in Figure 4.

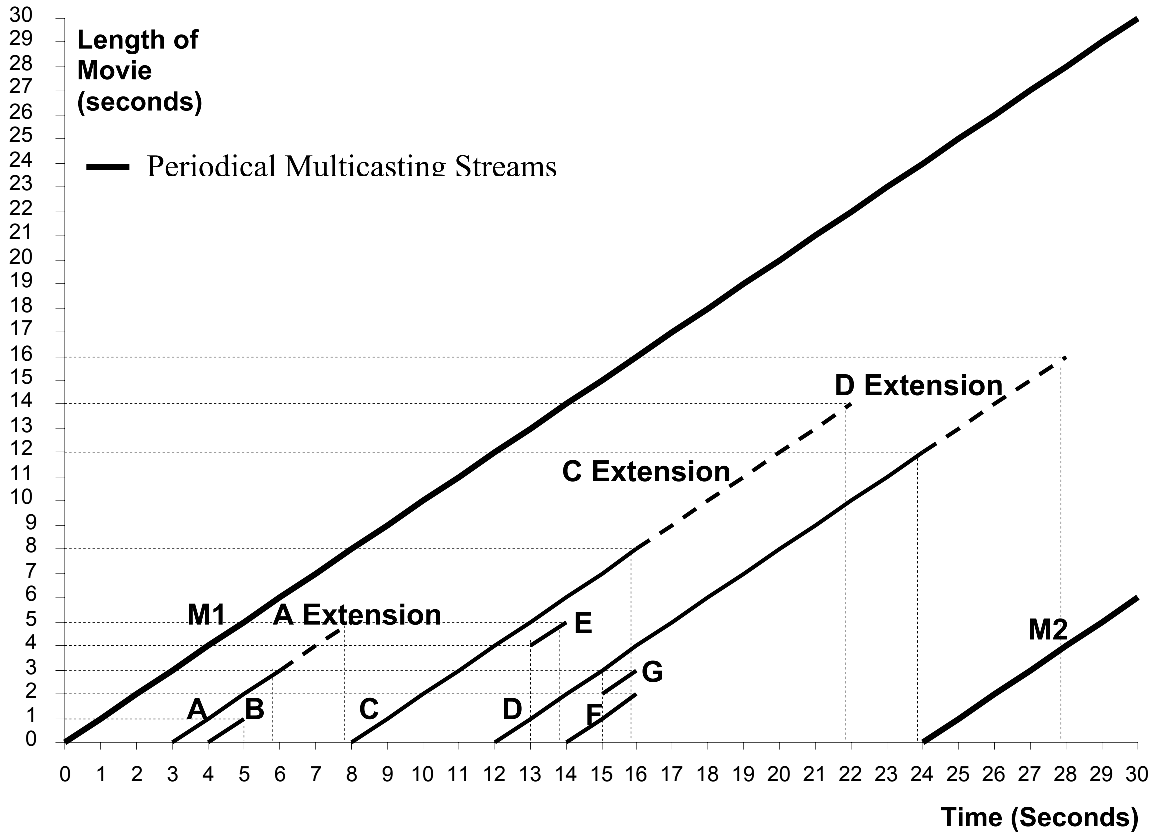

Figure 4: Best-effort Patching (BEP) Example 1

As shown in Figure 4, every 24 seconds a periodical multicasting stream is launched by the server. Both M1 and M2 are periodical multicasting streams with a lifetime set to the length of the movie. At the 3rd second, the first client-initiated stream A starts. The closest periodical multicasting stream with earlier play-point is M1, which becomes the merge target of A in that no first layer streams are merging into M1. As the displacement between A and M1's play-points is 3 seconds, A's lifetime is set to 3 seconds.

Stream B begins at the 4th second. It also finds M1 as the targeting periodical multicasting stream. According to dynamic merging technique, BEP continues to search through first layer streams targeting M1 and finds stream A reachable. Stream A turns into stream B's merge target. As a result stream A's lifetime is extended until the end of

the 7th second. The reason for the extension is the same as explained in ERMT, example 2. Stream B is now a 2nd layer stream in the merge tree formed by A and B. Its lifetime is set equal to the play-point gap from A's.

Stream C is created by a client request at the 8th second. It locates the nearest periodical multicasting stream, M1. The only first-layer stream merging into M1 is stream A, which is not reachable. Therefore, stream C selects M1 as the merging target. Its lifetime is 8 seconds.

At the 12th second, stream D is initiated. After going through the same procedure, it chooses M1 as the merging target, as well. It is important to note that as a first layer stream aiming at M1, stream C is believed to be not reachable by stream D in terms of BEP's classification, although it is reachable by ERMT's definition. If stream D was merging into stream C, the merge would occur at the same time that stream C was supposed to end. Now targeting stream M1, stream D's lifetime is set to last until the end of the 23rd second.

Stream E clearly must be caused by some jump in VCR operations because its play-point starts at the 4th second of movie content. The assumption is that a new client wants to skip the first four seconds of the movie. This researcher will illustrate later in this paper that such operation is not allowed, as clients must start from the play state with the first second of the movie before they can select any VCR operations. Here stream E and stream G, later on, are merely brought into play in order to illustrate the details of BEP. Stream C is the only reachable first layer stream for stream E. Stream C's lifetime is expanded to the end of the 21st second as a consequence of letting stream E merge. Together, streams C and E outline the second merge tree aiming M1.

For stream F, arriving at the 14th second, there are two reachable streams among the total of the three first layer streams, streams A, C and D, which are merging into stream M1. To determine which of the reachable streams, C or D, to pick as merge target of stream F, some calculations are required, according to the 2nd part of the dynamic merge technique.

Saving sC - chooses stream C as merge target of stream F:

sC1 = saving of merge if lifetime extension of stream C is unnecessary

= play-point displacement stream C to stream M1

= 8 seconds

sC2 = saving of merge if lifetime extension of stream C is necessary

= stream C's lifetime left after the stream F merges in

= ending time of stream C – expected merge time of stream F

= ending time of stream C – (play-point displacement between stream C and F + current time)

= 22nd second – (6 seconds + 14th seconds)

= 2 seconds

sC = minimum (sC1, sC2)

= 2 seconds

Saving sD - chooses stream D as merge target of stream F:

sD1 = saving of merge if lifetime extension of stream D is unnecessary

= play-point displacement stream D to stream M1

= 12 seconds

sD2 = saving of merge if lifetime extension of stream D is necessary

= stream D's lifetime left after the stream F merges in

= ending time of stream D – expected merge time of stream F

= ending time of stream D – (play-point displacement between stream D and F + current time)

= 24th second – (2 seconds + 14th seconds)

= 8 seconds

sD = minimum(sD1, sD2)

= 8 seconds

As illustrated by the calculation above, choosing stream D as stream E's merge target saves more stream time. As a result, stream D's lifetime is extended to the end of the 27th second. Stream G is another stream triggered by jump VCR operations. It also targets stream D for merge after a similar calculation to stream F's is done. Since it will merge into stream D at the same time as stream F will, choosing stream D as merge target will not cause a further lifetime extension of stream D.

Even though simulations and analysis show a significant performance gain of BEP over conventional VOD protocols, there is room for improvement. First, due to the restraint of patching in nature, only two-layer merging trees are formed. Second, the authors of paper [13] suggest that the interval of periodical multicasting can be adjusted according to a media file's request rate, but it fails to provide a way to find the optimal interval size.

In view of the fact that both ERMT and BEP are stream merging techniques (patching is a one-layer stream merging), the following common rule of merging technique is always valid to them: if the play-point displacement difference between a stream S and a candidate merge target is bigger than any of S's clients' local buffer can handle or longer than half the movie duration, then merging is impossible. Another merge target should be selected; otherwise, stream S will extend its lifetime to the end of the movie.

3. Client Interactivities

VCR operations are caused by VOD clients' interactivities when they are accessing video files such as movies. Common VCR operations include play, resume, stop, pause, slow motion, jump, fast forward, and rewind. These operations can be categorized in many different ways. The first is to group them by the moving direction relative to regular play direction. Play, resume, slow motion forward, jump forward, and fast forward are forward operations. Stop and pause are idle operations. Jump backward, rewind, and slow motion backward belong to backward operations. From a VOD server support point of view, VCR operations can be divided into streaming dependent and streaming independent. Steaming independent VCR operations include pause and stop operations that do not require a VOD server to provide video content to clients. All other VCR operations are more or less streaming dependent.

From the clients' angle, VCR operations consist of two major types: VCR operations with and VCR operations without video content displayed. Play, slow motion, fast forward, and rewind operations require constant video content display. Stop and jump operations do not have to show any pictures to the clients. Pause is a special case. Even

though an image is presented to clients, it never changes. This image is merely a leftover from a previous operation. Thus pause is also considered a VCR operation without video content displayed.

From another client viewpoint, VCR operations can be divided into the play operation and other operations in which play is the most fundamental feature and normally set as the default operation. That is why, on most occasions, play is excluded when the term “VCR operations” is mentioned.

The conventional technique to support VCR operations is straightforward. Each client requesting VCR operations is assigned a dedicated stream channel. This technique guarantees full support of any VCR operations. The drawback of this method is obvious. It requires a huge amount of resources to satisfy a relatively small number of clients. This disadvantage was not a problem until resource-sharing techniques were introduced. The resource-sharing techniques rely on predictable client activities that are most likely simple play operations. Resources such as server hardware and network between clients and server can be shared efficiently only when relations among clients’ play-points are relativity fixed. VCR operations break these relations. The more VCR operations that are executed, the less resources can be shared. For this reason many VOD servers that employ resource-sharing techniques are not able to offer full VCR support. However, VCR operations are a highly desirable feature today. True VOD means that it is able to satisfy any of clients’ VCR operation requests instantly.

Although VCR operations deeply impact the VOD’s performance, the effects are different among different resource-sharing techniques. Stream merging is one of the most efficient resource-sharing techniques; ERMT is the stream-merging technique with the

closest cost to the optimal merge tree [10, 31, 32]. Adding VCR support to ERMT is the major goal of this project, as well as finding out the VCR support's efficiency through simulations. BEP is a technique specifically designed for true VOD in order to handle VCR requests better [13]. BEP's performance makes excellent criterion, which can be used to evaluate the performance of ERMT with VCR support.

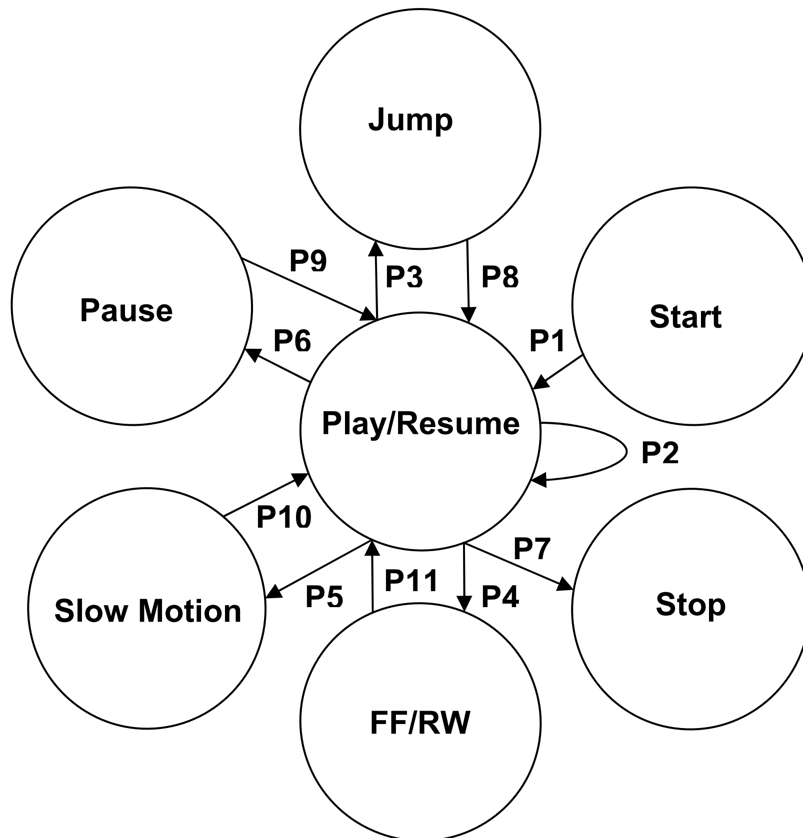

Figure 5: State Machine of User Activity

User VCR operations can be modeled by a state machine, as shown in Figure 5. There are seven states: start, play/resume, pause, jump, fast-forward (FF) and rewind (RW), slow motion, and stop. The jump state includes jumps of both directions—forward and

backward. Slow motion can be only forward in this model. The stop state can be reached in two cases. One case is triggered by the user, and the other occurs when the end of a movie is reached. Other VCR user operation states are straightforward, as illustrated in Figure 5.

There are several assumptions used in this model that require some explanation. First, a client is not able to initiate any VCR operations without going through the play state first after joining the server. This was mentioned in the previous section's BEP example 2. Furthermore, from all VCR operations, other than play/resume and stop, one can move only to the play/resume state. There are no interconnections between these operations. For example, one cannot pause, jump, or stop during a fast forward, rewind, or slow motion operation. To exit, clients must be in play mode first. It also is not possible to switch from slow motion to fast forward or rewind by bypassing the play operation. As mentioned earlier, VCR operations are generated by user interactivities. In addition, user interactivities also decide how frequently users move among these VCR operations. Transition probabilities among these states are provided in Table 1 and Table 2, based on some previous work [2, 13, 30]. To test VOD servers' performances under user interactivities, these transition probabilities will be used later as input values for a workload parameter in simulations.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
Probability	1	0.46	0.16	0.16	0.04	0.08	0.1	1	1	1	1

Table 1: Transition Probabilities among VCR Operation States

From	Play /Resume	Jump		VCR with Display			Pause	Stop	Sum
		Jump F	Jump B	FF	RW	Slow Motion			
Play	0.46	0.08	0.08	0.08	0.08	0.04	0.08	0.1	1
Jump	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
Pause	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
Picture VCR	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1

Table 2: Detailed Transition Probabilities among VCR Operation States

The transition probabilities of P1 through P11 marked in Figure 5 are shown in Table 1. It is important to note that P3 and P4 include the sub-possibilities of moving in both forward and backward directions. The assumption here is that there are equal chances of moving both directions. Therefore, P3 is split 50/50 between the possibilities of jumping forward or jumping backward. P4 is split in a similar way for fast forward and rewind. The details can be found in Table 2. To present this information more conveniently, other transition possibilities while moving among different VCR operations are also listed in Table 2. The first column of Table 2 is for departure states. The following columns represent destination states. The last column is for the sums of all transition possibilities starting from departure states. It is apparent that the values of this column must be all 1s. Table 2 is merely a more detailed version of Table 1, represented in a two-dimensional fashion.

State		Average Duration(minutes)
Play		10
Jump		0
Pause		5
VCR with Display	FF/RW	2.5
	Slow Motion	2

Table 3: Durations of VCR States

Another aspect of user interactivities is the average duration that users spend at each VCR state before moving to the next state. Table 3 contains the duration data. The durations of play, pause, and VCR with display operations will be used in the simulations as input of an exponential function. The duration of jump is always 0.

State		Speed
Play		1
Pause		0
VCR with Display	FF/RW	3/-3
	Slow Motion	0.5

Table 4: Speeds of VCR States

Table 4 defines the relative speed to play operation, whose speed is defined as 1. These speeds will be used in the simulations as well.

4. Proposed Algorithm

No user interactivities information is mentioned in the ERMT papers [10, 31, 32] studied for this project, nor is any ERMT VCR support found in the algorithm described in any ERMT papers. In the following section, an algorithm is proposed to enable VCR operations support for ERMT technique. As stated in the first section of this paper, several methods are available to provide VCR operations on VOD servers equipped with resource-sharing techniques, including contingency pool [14, 15, 16, 17, 18, 19], client local buffer [13, 24, 25, 26, 27, 28], and periodical multicasting [13, 26]. The proposed algorithm will utilize both client local buffer and periodical multicasting. The contingency pool technique is to reserve a certain number of channels for VCR support later. This technique is based on statistics and studies of a specific server's user admission and VCR request rate. Since admission control is beyond the scope of this project, the contingency pool technique is not employed in the proposed algorithm. Another traditional VCR support technique is introduced in previous section of this paper. It is to assign a dedicated channel for VCR request. The proposed algorithm also makes use of this method for some portion of VCR operations with continuing displays.

4.1 Client Local Buffer Utilization

ERMT technique requires client local buffer even when ERMT is not required. Thus using local buffer to support VCR operations is not only possible but also natural for ERMT. The assumption for both BEP and ERMT is that clients should have the minimum local buffer required by the resource-sharing techniques. The buffer can be utilized in two different ways for VCR operation support. The first is to save incoming

video content for later usage. When a client is at play, pause, or VCR with display state, as long as the client is receiving from one or two streams, the video contents should be saved to the local buffer as much as possible.

Due to the size limit of the client local buffer, different parts of the video relevant to current play-point are assigned with different priorities, as follows: already displayed content is set to a lower priority than not-yet-displayed video content; and the oldest content of already displayed content has the lowest priority and will be overwritten when buffer limit is reached. Among the not-yet-displayed video contents, the closer the content is towards current play-point (or pause-point), the higher the priority of the content. For example, for a client at pause state, buffering should be stopped when the local buffer is full instead of overwriting the oldest content that is later than the pause-point.

Another way of client local buffer utilization is to use it as much as possible before requesting any video contents from VOD server. This method is applicable to VCR operations with continuous display and to resume operations following pause, jump, and VCR with continuous display states. A local client buffer should always be checked first, no matter if incoming video content exists or not. If the requested play-point is found, the operation should be kept on the client local buffer as long as it is achievable. Details of client local buffer utilization can be found in section 4.3.

4.2 Periodical Multicasting

Fixed-interval periodical multicasting technique distributes multicasting streams evenly across the timeline. It increases the chances that any VCR operation requests find

a reachable multicasting stream nearby. Therefore, the proposed algorithm includes this technique, as well. The interval size of fixed-interval periodical multicasting is related intuitively to the popularity of the video content on the VOD server. If periodical multicasting streams are launched too frequently while admission rate and client interactivity degrees are low, then these multicasting streams will be wasted. In contrast, increased intervals of periodical multicasting may initiate too few multicasting streams, which are not able to fulfill the need of VCR operation requests. The authors of paper [13] suggested regulating the interval of periodical multicasting according to a media file's request rate, which depends on the file's popularity but did not offer a way of deciding the best possible interval size. The authors of paper [26] provided the following method for working out the right frequency of multicasting.

Formula for Calculating Fixed Multicasting Interval Size

$$x = 2T / \sqrt{2M} \qquad [26]$$

x: the interval size for a video in terms of time

T and M: over a period of time T, there are M requests for the video on average.

Thus, the multicasting-interval size of a video is related to its popularity. One factor that might be missing in this method is degree of user interactivities if the requests number is exclusively depending on the interarrival rate of initial admission of a movie or video file. VCR operations generate only more requests for the same movie. The requests most likely do not demand play-point from the beginning of the movie either. This method will be verified by later simulations.

4.3 Algorithm in Details

Utilizing client local buffer and initializing fixed periodical multicasting streams are simply methods to improve VOD performance in VCR request support circumstances. They are important add-ons, but only if fundamental VCR support is already in place. The essential component required to enhance ERMT with VCR operations support is how to process each of these VCR operations properly. Every transition marked in Figure 5 must be precisely managed in the proposed algorithm. What follows is the comprehensive description of the algorithm.

4.3.1 The proposed algorithm is an event-driven algorithm. Corresponding actions are triggered when events take place.

4.3.2 Types of Events

Events are categorized into two groups: events sent from clients to server, or events scheduled by the server.

- Events from client to server
    - Start
    - VCR with continuous display (FF, RW, and Slow Motion)
    - Resume
    - Stop

- Events scheduled at the server

- Merge
- Launching multicasting channels with fixed interval

4.3.3 Assumptions

- A client's buffer size is large enough (i.e., ½ of the movie length) to bridge between the current channel and the target channel.
- There are an infinite number of channels. The goal of the experiment is to measure the salability of this algorithm. That is, the number of server channels required to support the given number of concurrent clients with a certain degree of user interactivity will be measured. For example, the number of required server channels increases linearly (or logarithmically, etc.) to the arrival rate of clients.
- Later, future work of this project includes developing some intelligent admission control policy to evaluate how this policy can reduce the rejection probability for the given system and workload parameters.
- After a VCR operation, a client always goes to the play/resume mode.

4.3.4 Terminology

- A channel is used to multicast video data to multiple clients.
- A client can receive data from three different channels:
  - Current channel: a channel that multicasts video data at current playback point of the client for immediate playback service.
  - Target channel: a channel that multicasts the later part of video data than the current playback point of this client.

- I channel: a channel to support an individual client for VCR with Continuous Display operation.

- Client local buffer has two different usages:

- Current buffer: to buffer data from the current channel.
- Target buffer: to buffer data from the target channel to bridge between the current channel and target channel.

4.3.5 Server Model

- The server handles following events from clients:

- Start
  - A new client request is first queued in the corresponding queue based on its video ID.
  - When a channel, $channel_{current}$, becomes available, the server selects one of the video objects, such as x, using a scheduling policy such as FCFS, MQL, or MFQ. The channel is the current channel that immediately serves the batch of clients requesting the video x.
  - The server also looks for a target channel, $channel_{target}$, for the current channel to merge using the Find (start time is current time; play point is the beginning of the movie) subroutine.
  - If there is no target channel due to the fact that no preceding batch of clients is watching this movie:
    - Register clients (ID) with the $channel_{current}$.
    - Send a start response ($channel_{current}$, null) to all the clients in the batch.

- Start multicasting video data to the clients at $channel_{current}$.

- If a target channel is found:
  - Register the client (ID) with the $channel_{current}$
  - Register the client (ID) with the $channel_{target}$
  - Send a start response ($channel_{current,}$ $channel_{target}$) to the all the clients in the batch.
  - Start multicasting video data to the clients at $channel_{current}$. (The target channel is already multicasting video data. This new batch of clients simply joins the existing multicasting channel.)

➢ VCR with Continuous Display (FF, RW and Slow Motion)

- Receives "Give me channel i" request from a client:
  - Sends a response assigning $channel_i$ to the client.

➢ Resume

Case 1: Regular request from client:

- Assigns a $channel_{current}$ right away; the channel is the current channel that immediately serves the clients requesting the video from the resume point.
- The server also looks for a target channel, $channel_{target}$, for the current channel to merge using the Find (start time is current time; play point is resume play point) subroutine.
- If there is no target channel because no preceding batch of clients is watching this movie:
  - Register client (ID) with the $channel_{current}$.
  - Send a start response ($channel_{current}$, null) to the client.

- Start multicasting video data to the client at $channel_{current}$.

- If a target channel is found:
  - Register the client (ID) with the $channel_{current}$.
  - Register the client (ID) with the $channel_{target}$.
  - Send a start response ($channel_{current}$, $channel_{target}$) to the clients.
  - Start multicasting video data to the client at $channel_{current}$. (The target channel is already multicasting video data. This new client simply joins the existing multicasting channel.)

Case 2: Special request from client:

- Separate the client from the $channel_{current}$. If the client was the last one, release the $channel_{current}$ to the pool.
- Register the client (ID) with the original $channel_{target}$, which becomes the current $channel_{current}$ for the client.
- If there is no target channel due to the fact that no preceding batch of clients is watching this movie:
  - Register client (ID) with the $channel_{current}$.
  - Send a start response ($channel_{current}$, null) to the client.
  - Start multicasting video data to the client at $channel_{current}$.
- If a target channel is found:
  - Register the client (ID) with the $channel_{current}$.
  - Register the client (ID) with the $channel_{target}$.
  - Send a start response ($channe_{lcurrent}$, $channel_{target}$) to the clients.

- Start multicasting video data to the client at channel$_{current}$. (The target channel is already multicasting video data. This new client simply joins the existing multicasting channel.)

➢ Stop

- Separate the client from the current channel (if it exists) and from the target channel (if it exists), or from the I channel (if it exists).
- Release the channel(s) if they are not referred by any clients.

- Server handles the scheduled Merge event:

➢ Release the channel$_{current}$ (merger) from the batch of clients.

➢ If the target channel exists, register the batch of clients with the target (mergee) channel and mergee channel's target channel (if it exists) so that the clients in the batch are from now on receiving data from the mergee channel and the mergee's target channel.

- Server launches broadcasting channels with fixed interval:

➢ At a scheduled time (fixed intervals may vary from video to video), launches a channel to multicast the video from the first frame.

- Find subroutine:

➢ Server's Find subroutine using ERMT:

Step 1. Set = {non-I channel x | x's play point > given channel's play point}
If start time < = current time, go to step 2; otherwise go to step 3.

Step 2. Target channel = minimum$_{element \in Set}$ {gap = element's play point - given channel's play point | (current time + gap) < = element's scheduled ending time} if target channel is NULL, go to step 4; otherwise do following:

- gap = target channel's play point - given channel's play point; if gap > minimum (movie's duration/2, client local buffer size) go to step 4; otherwise: given channel's ending time = current time + gap; Use Find subroutine (ERMT algorithm) to find a target channel for target channel, start time = given channel's ending time.
- return target channel.

Step 3. Target channel = $\text{minimum}_{\text{element} \in \text{Set}}$ {gap = element's play point - given channel's play point | (start time + gap) < = element's scheduled ending time} if target channel is NULL, go to step 4; otherwise do following:

- gap = target channel's play point - given channel's play point; if gap > minimum (movie's duration/2, client local buffer size) go to step 4; otherwise: given channel's ending time = start time + gap; Use Find subroutine (ERMT algorithm) to find a target channel for target channel, start time = given channel's ending time.
- return target channel.

Step 4. Given channel's ending time = current time + length of the movie – given channel's play point; return NULL.

4.3.6 Client Model

A client is able to generate following events.

- Start
    - Send a start request.
    - Receive a "start response" from the server.

- Take an action according to the response from the server:
  - Case of ($channel_{current}$, null):
    - Receive and play data from $channel_{current}$ and store it in the current buffer.
  - Case of ($channel_{current}$, $channel_{target}$):
    - Receive and play data from $channel_{current}$ and store it in the current buffer.
    - Receive data from $channel_{target}$ and store it in the target buffer.

- VCR with Continuous Display (FF, RW and Slow Motion)

- The operation is first served from the local buffer, if possible. Only if there are no available data in the local buffer, or local buffer no longer supports: the client sends "Stop" request to the server and generates "Start VCR with Continuous Display" request to the server.
  - Receive a VCR with Continuous Display response ($channel_i$) from the server.
  - Start receiving and playing video data from the $channel_i$.
  - If the end of movie is reached, send "Stop" request to server.

- VCR without Continuous Display (pause/jump)

- Case of pause:
  - Receive data from $channel_{current}$ (if exists) and $channel_{target}$ (if exists) and store it in the current buffer and target buffer.
  - Send "Stop" request to the server when buffers are full.
- Case of jump:
  - Trigger Client Resume event with a play point.

- Resume

- Case 1: Resume from Jump:

  Step 1. If found resume point in local buffer do following, otherwise go to Step 2:

  - If it was jump backwards and has $channel_{current}$, play from local buffer, continue receiving from $channel_{current}$ and store it in current buffer; continue receiving from $channel_{target}$ (if exists) and store it in target buffer.
  - If it was jump forward and was receiving from both $channel_{current}$ and $channel_{target}$, play from local buffer and send Special resume request to server. Take an action in accordance to the response from the server:
    - Case of ($channel_{current}$, null):
      - Receive data from $channel_{current}$ and store it in the current buffer.
    - Case of ($channel_{current}$, $channel_{target}$):
      - Receive data from $channel_{current}$ and store it in the current buffer.
      - Receive data from $channel_{target}$ and store it in the target buffer.
  - If it is not the two cases above, send "Stop" request to server (in case the client was registered with any channel) and play on buffer. When buffer can no longer support, send resume request to server, and take an action according to the response from the server.
    - Case of ($channel_{current}$, null):
      - Receive and play data from $channel_{current}$ and store it in the current buffer.

- Case of ($channel_{current,}$ $channel_{target}$):
  - Receive and play data from $channel_{current}$ and store it in the current buffer.
  - Receive data from $channel_{target}$ and store it in the target buffer.

Step 2. Cannot find resume point in local buffer:

- Send "Stop" request to the server. Send resume request to server. Take an action according to the response from the server:
  - Case of ($channel_{current}$, null):
    - Receive and play data from $channel_{current}$ and store it in the current buffer.
  - Case of ($channel_{current,}$ $channel_{target}$):
    - Receive and play data from $channel_{current}$ and store it in the current buffer.
    - Receive data from $channel_{target}$ and store it in the target buffer.

➢ Case 2: Resume from Pause

- If the client is still receiving from $channel_{current}$, play from local buffer, continue receiving data from $channel_{current}$, and store it in current buffer; continue receiving data from $channel_{target}$ (if exists) and store it in target buffer.
- If client no longer receives from any channel, local buffer is supposed to be full; play back from local buffer. When local buffer no longer supports, send

Resume request to server. Take an action according to the response from the server.

- Case of ($channel_{current}$, null):
  - Receive and play data from $channel_{current}$ and store it in the current buffer.
- Case of ($channel_{current}$, $channel_{target}$):
  - Receive and play data from $channel_{current}$ and store it in the current buffer.
  - Receive data from $channel_{target}$ and store it in the target buffer.

➢ Case 3: Resume from VCR with Continuous Display (FF, RW and Slow Motion)

Step 1. If resume play point is found in local buffer, do the following; otherwise go to Step 2.

- If the client is not registered with any channel, play from buffer until local buffer no longer supports, then send Resume request to server. Take an action according to the response from the server.
  - Case of ($channel_{current}$, null):
    - Receive and play data from $channel_{current}$ and store it in the current buffer.
  - Case of ($channel_{current}$, $channel_{target}$):
    - Receive and play data from $channel_{current}$ and store it in the current buffer.

- Receive data from $channel_{target}$ and store it in the target buffer.

- If the client was receiving from I channel, send "Stop" request, play from buffer until buffer no longer supports, then send Resume request to server. Take an action according to the response from the server.
  - Case of ($channel_{current}$, null):
    - Receive and play data from $channel_{current}$ and store it in the current buffer.
  - Case of ($channel_{current,}$ $channel_{target}$):
    - Receive and play data from $channel_{current}$ and store it in the current buffer.
    - Receive data from $channel_{target}$ and store it in the target buffer.

Step 2. Resume point not found in local buffer.

- Send "Stop" request (no matter whether the client was receiving data from any kind of channels); send Resume request to server. Take an action according to the response from the server.
  - Case of ($channel_{current}$, null):
    - Receive and play data from $channel_{current}$ and store it in the current buffer.

- Case of ($channel_{current,}$ $channel_{target}$):
  - Receive and play data from $channel_{current}$ and store it in the current buffer.
  - Receive data from $channel_{target}$ and store it in the target buffer.

- Stop
  - Send "Stop" request to the server.
  - Client exit.

4.4 Implementation – Hierarchical-Stream Merging and Stream Multicasting with Fixed Interval

4.4.1 Introduction of CSIM 19

CSIM 19 [33] is a product of Mesquite Software, Inc. It includes a C or C++ library with many functions. This library can be used to build "process-oriented, discrete-event" [33] computer models to simulate complicated systems such as VOD service. CSIM 19 is widely used in VOD simulations around world. The main components introduced by CSIM 19 are facility and process. Facility is used to model a resource such as a store, a PC, or a VOD server. Process is for modeling dynamic entities such as customers, network packages, or multicasting video streams. Facility is not used in either the implementation of ERMT or that of BEP since the VOD server's usage is not the concern of this project. The goal is to discover the number of video streams needed to support a certain number of admission and VCR requests. Both video streams and clients are

modeled by CSIM processes. The CSIM process has several advantages. CSIM is able to control a huge number of processes at the same time, according to simulation time. A process can switch an infinite number of times between computing and holding states. Holding means to pass simulation time; computing means to utilize computer time. Another reason CSIM 19 was selected for the implementation is that it has built-in functions, such as exponential() and prob(). These frequently used functions make implementation of simulation much easier. In addition, CSIM provides data tables for result recording and statistics. It also offers "automatic run-length control" [33], which helps decide simulation duration based on desired result precisions.

4.4.2 Data Structure

As shown in Figure 6, there are mainly three structures used in ERMT and BEP implementations: Movie, Client, and Stream. The movie array is formed by all movies provided by the VOD service. Each movie maintains a link list that is made of multicasting Streams of the movie. A multicasting stream is defined as a play-speed stream without merge target. A multicasting stream does not end until its play-point reaches the end of the movie. Each member of a multicasting Stream link list is the root of a stream merging tree. A Stream is received by 0 to many Clients. It is equipped with two Client points that point to the first and last Client of a link list made up of Clients of this Stream. Each Stream also has four Stream pointers that are, respectively, for its merge target Stream, the first Stream of a list of Streams targeting it, its previous Stream, and the next Stream in the list of Streams targeting its merge target. Type is one data field of Stream structure. A Stream could be one of following four types: fixed-interval

multicasting type, other multicasting type, patching type, and dedicated VCR support type. The first two types are the roots of the merge trees. Patching types are Streams with merge target. Patching Streams make up the biggest group among these four files. A dedicated VCR support type Stream does not have a merge target; it is used as for non-play state VCR operations support. Client structure has one Stream pointer and two Client points. The Stream pointer is for the Stream from which it receives patching video content. If the Stream has a merge target, the Client also receives video content from the target Stream. The two Client pointers are for a Client's previous and next sibling Clients that are referring the same Stream.

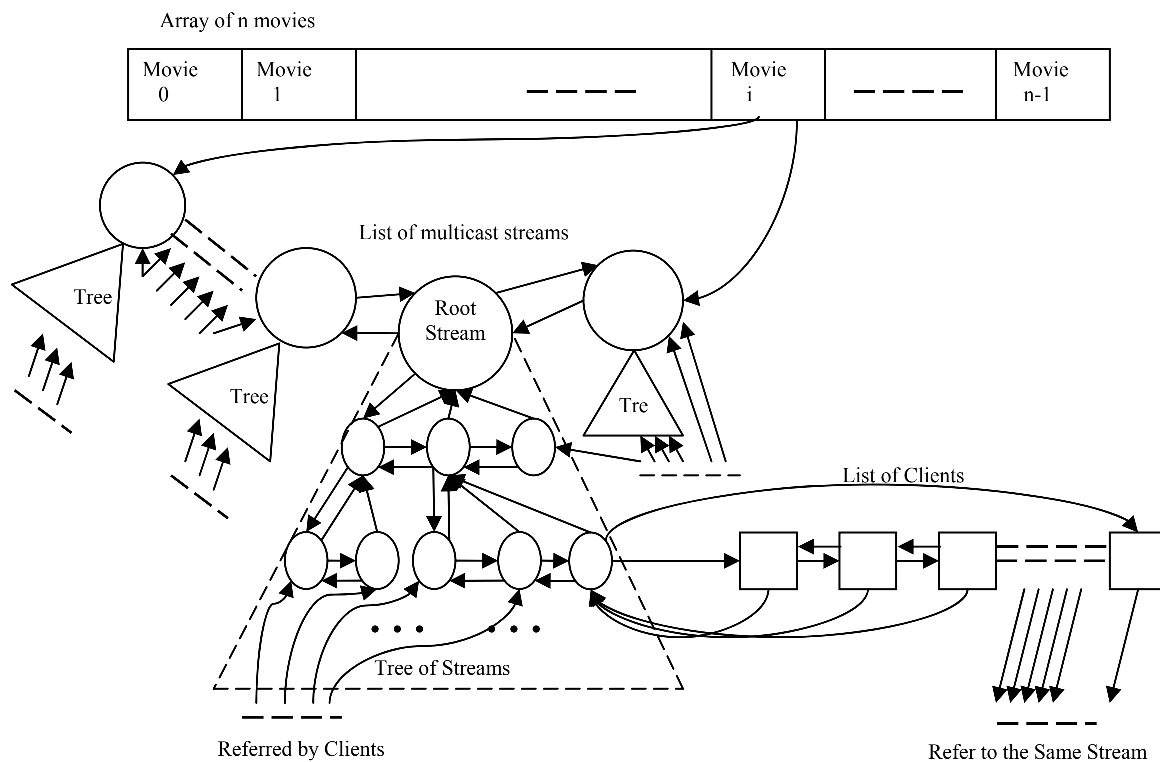

Figure 6: Data Structure: ERMT and BEP

4.4.3 Implementation Remarks

Implementation took the longest time of this project, owing to the complexity of the algorithms and VCR operations handling. The entire implementation for both ERMT and BEP is written in C language on Standard Microsoft Visual Studio 6.0 with CSIM 19's library for C used.

4.4.3.1 ERMT with VCR Support

Transition possibilities among different VCR states are implemented using prob() functions. Based on data from Tables 1 and 2, a VCR operation is picked. Play-points of jump operations are put into practice with prob() function as well. For jumping forward, a play-point is selected by multiplying the movie time left with prob()'s return value, which is always between 0 and 1. Therefore, this jump-point picked must be between current play-point and the end of the movie. By the same token, a jump-point between the beginning of the movie and current play-point is randomly chosen for jump backward.

Exponential function is used to decide the time of next admission request taking an inter-arrival rate as the input parameter. Durations for VCR operations are also selected by utilizing exponential() functions. For each VCR operation, the corresponding duration found in Table 3 is taken as input parameter, the returned value of exponential() is then set as the duration of the VCR operations. Movie popularities follow zipf distributions.

In order to discover the improvements that multicasting and client local buffer technique makes on VCR support, fixed-interval periodical multicasting and local buffer support are set to optional in the ERMT implementation. In order to verify Little's law [1], VCR request simulation is also set to optional. These options can be turned on and

off by the use of three switches, respectively. Without fixed interval periodical broadcasting, the list of multicast streams (list containing roots of merge trees shown in Figure 6) may contain many fewer nodes. ERMT intends to build one big merge tree instead of many small trees as long as client buffer size and movie length allows. Even for an existing multicasting stream, ERMT tries to find a merge target for it from time to time. In the implementation, if fixed-interval periodical streams are deployed, they never merge into any other streams, whereas other multicasting streams are still subject to finding merge targets. Although these implementations were originally developed for ERMT, they are also deployed in the BEP VCR request simulation.

4.4.3.2 BEP

The BEP technique appears similar to ERMT from an implementation point of view. They both utilize client buffer and fixed periodical multicasting stream. In addition, they both build merge trees in between multicasting streams. However, they are very different. First, fixed periodical multicasting stream is optional to an ERMT algorithm. In section 5, ERMT both with and without periodical multicasting are simulated and analyzed. On the other hand, fixed periodical multicasting is a must for BEP due to its patching technique nature. Second, the maximum number of layers for a merge tree is two in BEP, whereas ERMT is able to build a merge tree with unlimited layers. Finally, the most significant difference is that ERMT and BEP have completely dissimilar algorithms for finding a merge target.

ERMT merge trees are frequently modified when merge targets change existing streams. Contrarily, streams in BEP do not switch the merge target once it is chosen. In

order to traverse through an ERMT merge tree, the merge target locating function is implemented in a recursive fashion. Additionally, a recursive update function is required. It is applied to each newly found merge target to perform a merge target search. BEP implementation employs a non-recursive target search function and does not require an update function.

There were a few technical difficulties in the implementation, too. For example, even though a detailed search method was provided in the BEP paper [13], the authors failed to point out a few minor details, such as how to handle VCR requests with play-points earlier than the first multicasting stream's. There could be two options: one is the conventional approach—always support this type of forward moving VCR operations with dedicated channels until the user exits; the other is to make the channel public multicasting once the user switches to play. After thorough evaluations, the second option was implemented because ERMT uses same method. This has become the principle of handling other unclear parts of BEP algorithm because BEP is brought into play purely for comparison purposes. These minor details have no significant impact on performance. As a matter of fact, by eliminating the irrelevant differences in the implementations, the simulation comparison becomes more meaningful.

During the implementation, a small typographical error was found in the BEP paper [13],which damages BEP performance, in that the lifetimes of some target streams are subject to unnecessary extension. Following is the pseudo code cited from [13]:

```
"struct MQueue {
  element *head; /*the head record*/
  int offset; /*the offset of queue*/
  int lifetime; /*the lifetime of queue*/
```

```
  int latime; /*joining time of the latest client*/
}
Algorithm DMA(Q, C, n, t)
/*Q is a set of merging queues*/
/*C is an interaction client*/
/*n is the group number*/
/*t is the arrival time of C*/
c = offset(C, n); /*get the offset of C in group n*/
qo.=max_{q∈Q}{min{q.offset + q.lifetime – c – (t – q.latime),q.offset} q.offset<= c < q.offset +
          q.lifetime – (t – q.latime)}
if (qo= null) {/* generate a new merging queue*/
  Generate(qo); /*generate a new queue*/
  qo.head is set to C; qo.offset = c;
 qo.lifetime = c; qo.latime = t;
}
else { /*merging C into qo */
   delta = t - qo.latime;
   qo.latime = t;
   lifetime(C) = c - qo.offset; /*the lifetime of patching stream C is changed */
   if(c > qo.lifetime - delta) qo.lifetime = c;
}”
```

As explained in Section 2 of this paper if a queue’s current lifetime is long enough to support C’s merge, there is no extension needed for the queue. The problem of the code is that the no-extension case is handled incorrectly. For example, suppose:

queue’s original $q_o.latime = 2$;

queue’s original $q_o.lifetime = 10$;

current time $t = 5$, which is also the arrival time of C

given c = 4;

in this example $c <= q_o.lifetime - delta$ in that $4 < 10 - (5-2)$. According to last line of the algorithm, there is no need to extend the lifetime of the queue; $q_o.lifetime$ still remains 10. However the latime of the queue, $q_o.latime$, is changed to 5 caused by the third latest line of the code, $q_o.latime = t$; Therefore, the queue will be released at time 15th second ($q_o.lifetime$ + new $q_o.latime$) instead of 12th second ($q_o.lifetime$ + original $q_o.latime$). The 3 second stream time is wasted due to the typographical error. So in the implementation, the last line of the algorithm is corrected into:

```
if(c > qo.lifetime - delta) qo.lifetime = c;
else qo.lifetime = qo.lifetime - t;
```

This modification is necessary because the algorithm is described clearly in paper [13]. The discrepancy of the pseudo code must be a mistake, which could distort simulation results.

5. Simulations and Analysis

5.1 Workload Parameters and Performance Metric

Workload parameters are input constraints used for running the simulations of this project. All simulations assume there is only one movie on the VOD server because the goal is to examine performance of proposed algorithm and to compare it with BEP performance in terms of VCR functionality. In the BEP paper, simulations are done with one movie on the server. This project's simulators can be extended to compare the performance of the algorithms for multiple movies. There are some factors to consider extending the simulators for this purpose such as the fact that there is only one interval

used for fixed-interval multicasting and the relation between movie popularity and optimal interval size for multicasting that is not known from prior studies. This researcher leaves the extension of simulation for multiple movies as future study. The movie's length is set to 90 minutes. Interarrival time of client requests is the interval (in seconds) between newly arrived client requests. The average interarrival time ranges between 6 and 60 seconds in the simulations. It is also based on paper [13]'s workload parameter for comparison purpose. Interarrival rate, so called λ, is the average number of requests coming to the server per minute. Therefore, λ is a reciprocal of interarrival time. Consequently λ's value varies from 1 to 10 per minute in the simulations. Fixed-interval of multicasting is the interval size set between multicasting streams launched by VOD server. The intervals used in the simulations of this project vary from 38 to 5400 seconds. Client buffer size represents the minimum client buffer space required by ERMT and BEP. The client buffer's unit size is a second, which stands for the space taken on the client local buffer in order to store video content of a second. Client buffer sizes range between 300 to 5400 seconds. VCR operation workload parameters are based on Table 1 through Table 4 of Section 3. Section 4.4.3.1 explains how these workload parameters actually are used. The basic workload parameters are listed in Table 5.

The ultimate goal of the simulations is to find how many streams are needed to satisfy a certain number of client requests. Therefore, the most representative performance metrics is the ratio of the current number of all streams to the current number of client requests being serviced. A lower ratio indicates better performance of the resource-sharing technique as fewer streams are required. The performance metric used in paper [13] is the ratio of the number of merging streams to the number of total streams. That

metric cannot be used for this project because it fails to reflect the relationship between the number of ongoing client sessions and the number of all existing streams.

Notation	Meaning	Value
λ	Arrival rate per minute to VCR server	1 ~ 10 /minute
w	Interval size of Fixed-interval multicasting	38 ~ 5400 seconds in normal play speed
d	Client local buffer size	300 ~ 5400 seconds in normal play speed
L	Length of the movie	5400 seconds in normal play speed
Pi	Transition probability between two VCR states	see Section 3 - Table 1
Di	Average duration of a VCR state	see Section 3 - Table 3
Si	Broadcasting speed of a VCR state	see Section 3 - Table 4

Table 5: Basic Workload Parameters

5.2 Simulations and Results

In this section, simulation results are illustrated and analyzed. Each data point given in the rest of Section 5 is an average result of a hundred runs of 10 hours (36,000 seconds) in terms of simulation time provided by CSIM 19. The performance of four different schemes is analyzed through simulations. They are BEP, M. ERMT (short for modified ERMT with VCR support and local client buffer utilization for VCR operations), M. ERMT without client buffer VCR support (short for modified ERMT with VCR support but without local client buffer utilization for VCR operations), and M. ERMT with MC (short for modified ERMT with VCR support, local client buffer utilization for VCR

operations, and fixed-interval periodical multicasting). The three modified ERMT-related simulations are to show the effect of local buffer and fixed-interval multicasting for VCR support. As mentioned in 5.1, the performance metric used in these simulations is the ratio of the number of streams to the number of user sessions. The ratios are comparable among these simulations because for each corresponding data result used, the absolute numbers of user sessions are almost identical among these simulations. The number of streams and current user sessions for each figure can be found in the Appendix.

5.2.1 Little's Law and Verification

Little's law is used for validation purposes in performance evaluations. It is described as follows:

Mean number in the system = arrival rate × mean response time [1]

$$N = \lambda \times t$$

Mean response time can also be referred to as mean time spent in the system. For example, the average number of customers entering a liquor store per minute is 5; each customer spends about 1 minute in the store. Hence λ is 5; t is 1; $N = 5 \times 1 = 5$. The mean number of customers in the store at any time is 5.

When VCR requests generating is turned off, each client session spends exactly the same amount of time as the length of the movie, meaning that each client watches the movie from the first frame to the last, which makes t equal to 90 minutes. Figure 7 illustrates the verification results of BEP, M. ERMT, and M. ERMT with MC simulations. This shows that the simulations are valid when there are no VCR requests generated since all the simulation results are equal to the calculation results according to Little's Law. As

shown in Figure 7, these simulation data points match the values of λ × t where t is the length of the movie. The periodical multicasting interval (for BEP and M. ERMT with MC) and client buffer size (for all) are both set to 300 seconds in these simulations. M. ERMT without client buffer VCR support is not shown in Figure 7 because it is identical to M. ERMT when clients are not VCR-interactive.

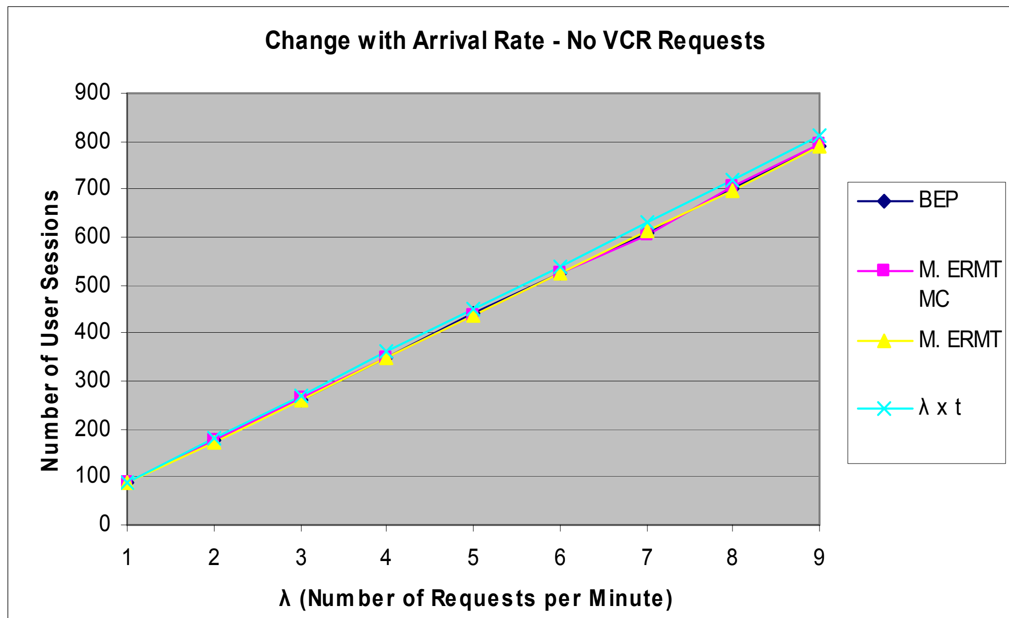

Figure 7 Validate Simulations Using Little's Law

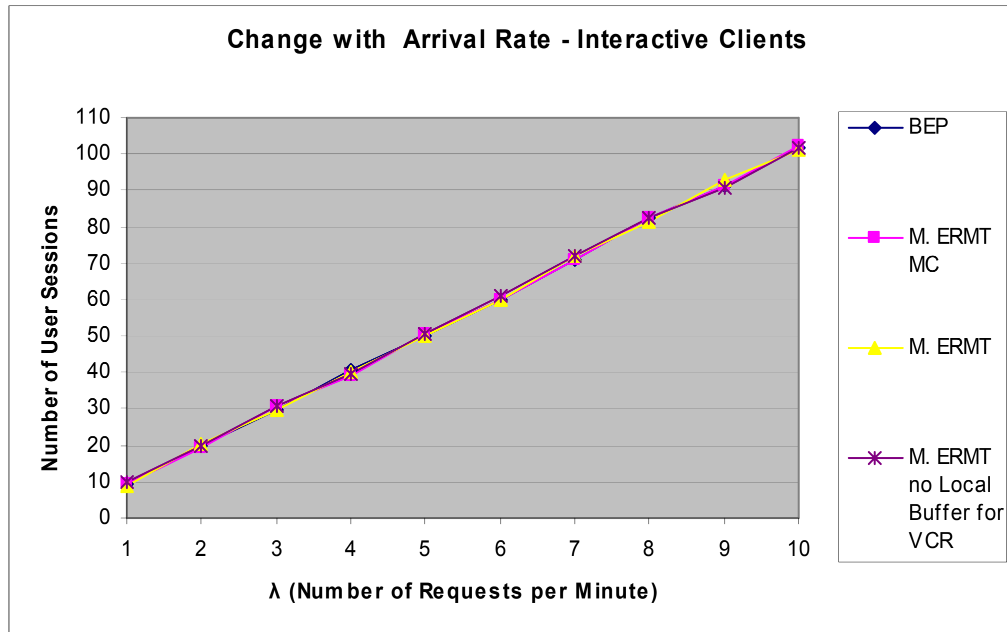

Figure 8 Number of User Sessions with VCR Requests—λ Changes

With VCR requests enabled, the average time that each session lives is unknown. However, given the results collected from simulations with VCR requests enabled, Little's Law can be used to deduct the mean time each user session takes. When users are interactive in terms of VCR operations, simulation results and Little's Law indicate that each user session lasts about 10 minutes according to the simulation results shown in Figure 8. Here is a sample calculation: $t = N / \lambda = 101.5 / 10 = 10.15$ minutes. Figure 8 shares the same input parameter values as Figure 7, except that VCR requests are enabled during simulations. It demonstrates that clients spend much less time (almost 80 minutes or 88.72% less in this case) on a video object when VCR operations are allowed.

5.2.2 Impact of Arrival Rate

This section represents how performance changes when arrival rate λ becomes larger.

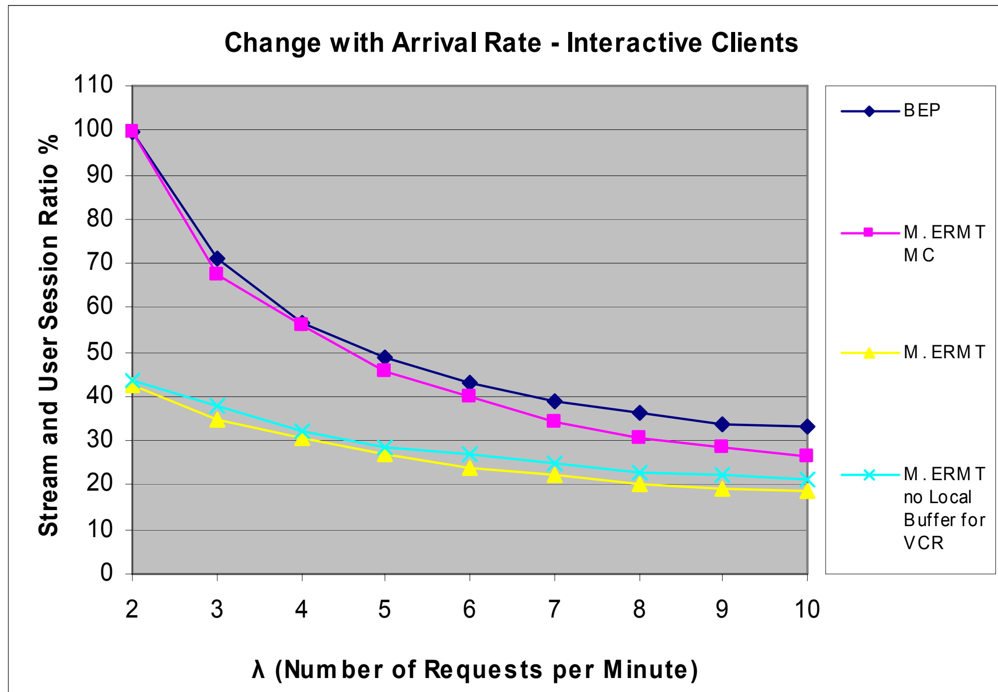

Figure 9 Performance under Interactivities—λ Changes

- Fixed input parameters other than the common ones of all simulation results shown in Figure 9:

  - Fixed-interval size of multicasting streams: 300 seconds in terms of play state. Note that M. ERMT and M. ERMT without client local buffer VCR support do not use fixed-interval multicasting, which is used only by BEP and M. ERMT with MC.
  - Client local buffer size: 300 seconds in terms of play state.
  - Clients are interactive in terms of VCR operations.

- Changing input parameter: interarrival time of client requests; it varies from 60 seconds to 6 seconds. Corresponding λ values (requests per minute) are 1 to 10.
- Observations of the result set:
    - For all four simulators, the result ratio goes down (performance improves) when λ's value goes up.
    - Among the simulators, M. ERMT has the lowest ratio, which means it has the best performance. Its ratios are at least 50% lower relative to BEP and M. ERMT with MC. BEP has the highest ratio, which means it has the worst performance. M. ERMT with MC's performance is in between but much closer to BEP. M. ERMT without local buffer support for VCR performs better than BEP and M. ERMT with MC but not as well as M. ERMT.

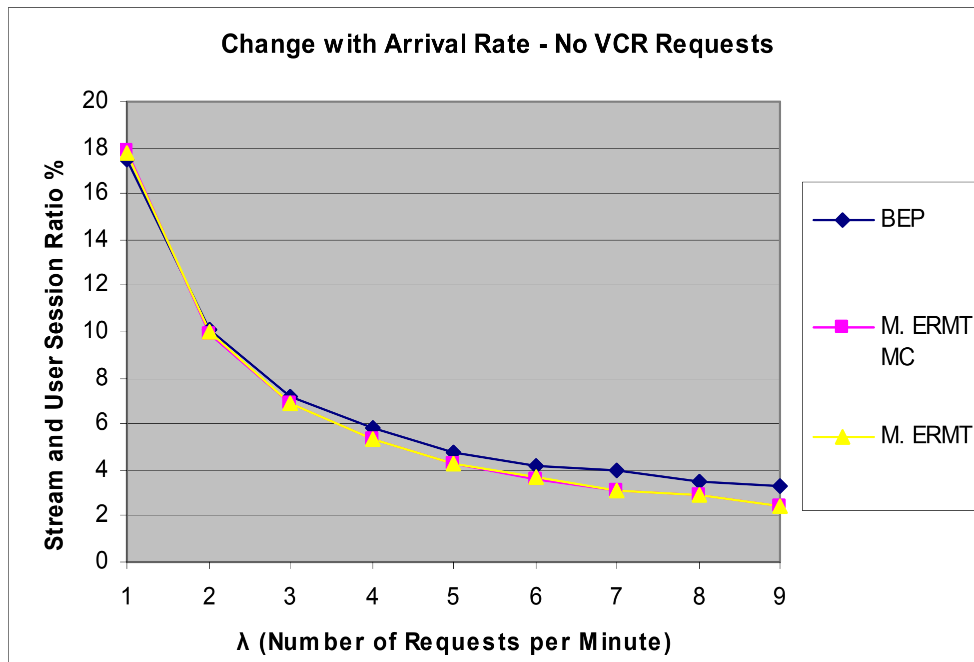

Figure 10 Performance without VCR Requests—λ Changes

- Input parameters for simulation results shown in Figure 10: same as previous data set except clients are not interactive. They always stay in play state until the end of the movie is reached.
- Observations of the result set: for all three simulators, the performance is much better than those illustrated in Figure 9. There are three simulations shown in Figure 10. M. ERMT without client local buffer VCR support's simulation is absent since it is identical to M. ERMT when clients are not VCR-interactive. The simulation results of BEP, M. ERMT, and M. ERMT with MC are very similar. This means that for given workload parameters, client requests can be satisfied with a similar amount of streams among BEP, M. ERMT, and M. ERMT with MC when no VCR requests are generated. M. ERMT and M. ERMT with MC perform almost identically for all the data points. They both slightly outperform BEP.

The simulation results above indicate that when other parameters are fixed, all simulations' performance improve when λ's value increases. The reason is straightforward—more clients cause more resource-sharing to take place. M. ERMT greatly outperformed all other techniques. BEP performs the worst. M. ERMT with local buffer support for VCR performs the best. M. ERMT without client local buffer support has better performance than M. ERMT with both client buffer support and fixed-interval multicasting. Fixed-interval multicasting costs system resources. The simulation results show that with the given workload parameters, this cost is greater than the benefit that

fixed-interval multicasting and client local buffer VCR support can provide, especially when the arrival rate is low. BEP and M. ERMT with MC's performance improve when the arrival rate increases because the multicasting streams are more utilized due to increased VCR requests. Without VCR operations, the three simulators perform almost identically.

5.2.3 Impact of Multicasting Interval and Client Buffer Size

It is very interesting to find how multicasting interval size and client buffer size affect all simulations.

5.2.3.1 Change Interval Size and Client Buffer Size Together

It is important to note that for the first four data points of BEP and M. ERMT with MC in Figure 11 below, multicasting interval sizes are 60, 120, 180, and 240 seconds ,while the client buffer sizes are all set to 300 seconds. For the rest of the data points of BEP and M. ERMT with MC, the multicasting interval size and client local buffer size have the same values. The reason the multicasting interval size and local client buffer size have to be changed together in these simulations for BEP and M. ERMT with MC is because the client local buffer size must be equal to or greater than the multicasting interval size to fully utilize the multicasting streams. In the cases of M. ERMT and M. ERMT without local client buffer VCR support, only client buffer size matters since there is no fixed-interval multicasting technique used. They are basically the same except for the fact that one supports VCR with local client buffer and the other does not. For M. ERMT and M. ERMT without local buffer support lines, shown in

Figure 12 below, the values of the x axis are client buffer size only. No interval size values are needed for M. ERMT and M. ERMT without local buffer support simulations since they do not require fixed-interval multicasting.

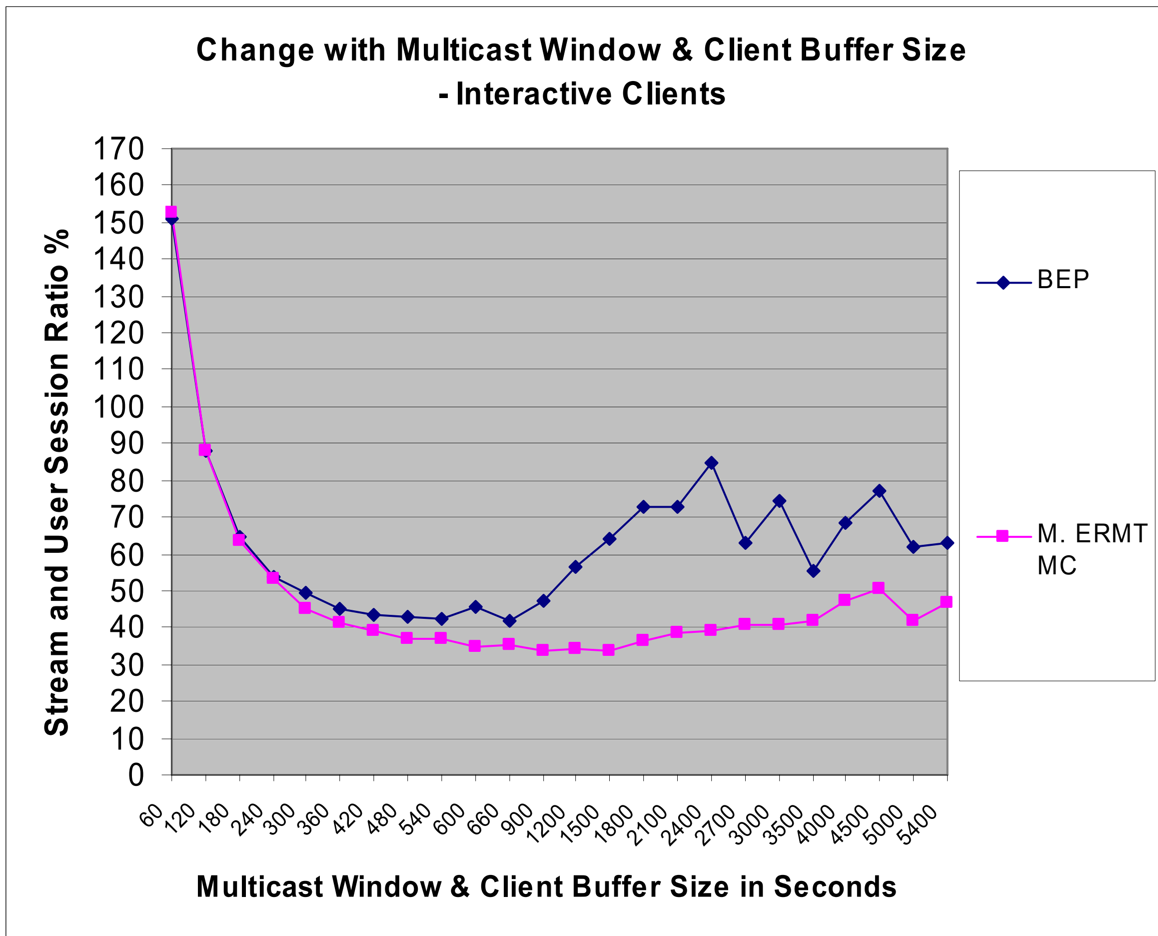

Figure 11 Performance with VCR Requests—Multicasting Interval and Client Buffer Size Change Together

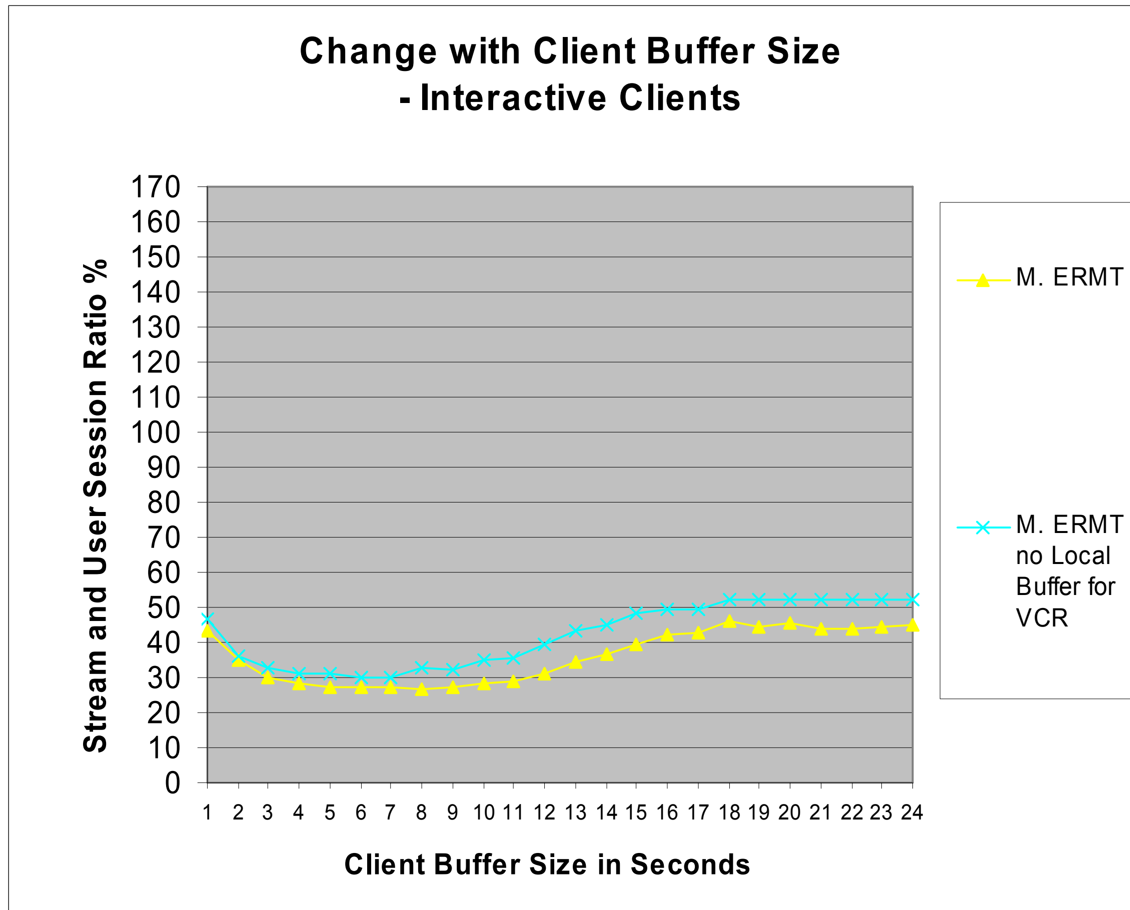

Figure 12 Performance with VCR Requests—Client Buffer Size Changes

- Fixed input parameters for all simulation results shown in Figure 11 and 12:
  - Interarrival time of client request is set to 12 seconds. Corresponding λ (requests per minute) value is 5.
  - Clients are interactive in terms of VCR operations.
- Changing input parameters:
  - w, Fixed-interval size of multicasting streams: varies from 60 seconds to 5400 seconds in terms of play state. Please note that M. ERMT and M. ERMT without client local buffer for VCR support do not require

fixed-interval multicasting, which is used by BEP and M. ERMT with MC.

- d, Client Buffer size: varies from 300 seconds to 5400 seconds for BEP and M. ERMT with MC (when w is smaller than 300 seconds, d is set to 300 seconds; when w is equal to or greater than 300 seconds, d is set to the same value of w); d varies from 60 seconds to 5400 seconds for M. ERMT and M. ERMT without client local buffer VCR support.

- Observations of the result set:
  - Among the simulators, M. ERMT has the lowest ratios, which means it has the best performance. Its ratios are at least 33% lower relative to M. ERMT with MC. BEP has the highest ratios, indicating that it has the worst performance. M. ERMT without local buffer VCR support performs worse than M. ERMT as expected. M. ERMT without local buffer VCR support performs better than M. ERMT with MC until w > 900 seconds. For both BEP and M. ERMT with MC, the ratios go down rapidly in the range of w = 60 to 300 seconds. BEP and M. ERMT with MC's poor performance with small multicast windows and client buffers are due to the fact that too many multicasting streams are generated and wasted. M. ERMT with MC's performance is between those of BEP and M. ERMT but closer to BEP's with a range of w = 60 to 900 seconds and closer to M. ERMT in the range of w > 900 seconds. BEP and M. ERMT with MC's ratios decrease

slowly in the range of w = 360 to 900 seconds. BEP performs the best when the multicasting window and client buffer size are set to 900 seconds. M. ERMT with MC's result ratio is the lowest when the multicasting window and client buffer size are set to 1200 seconds. These results indicate that there exist optimal multicasting interval sizes for BEP and M. ERMT with MC. BEP and M. ERMT with MC's result ratios increase when $w \geq 1200$ seconds because more VCR requests cannot be accommodated when fewer multicasting streams are generated due to the increased multicasting interval size.

- For M. ERMT, the line is much flatter. The ratios range between 26.7% (lowest when d = 480 seconds) and 46.05% (highest when d = 2700 seconds). It decreases when d ranges from 60 to 300 seconds and remains stable until it increases when d varies from 660 to 2700 seconds. It becomes stable again when $d > 2700$ seconds. Hence, the bigger client buffer size d does not always benefit the performance of M. ERMT even when clients are VCR interactive because when d is too big, patching streams are kept longer on the server. These simulations also show that there is an optimal client buffer size for M. ERMT for given workload parameter settings. The same analyses apply to M. ERMT without client local buffer VCR support as well. Without client local buffer supporting VCR, M. ERMT performance is definitely damaged by around 2 to 20%.

- M. ERMT with MC has slightly better performance than M. ERMT without client buffer VCR support and M. ERMT when w and d are greater than 900 and 1500 seconds, respectively.

- Analysis of the result set:
  - The simulation results do not show abnormal fluctuations among different resource-sharing techniques except for BEP, the performance of which loses a clear trend after the client buffer size becomes equal to or larger than 2700 seconds. The explanation is as follows. The basic rule of stream merging is: if the play-point displacement difference between a stream S and a candidate merge target is bigger than any of S's clients' local buffer can handle or longer than half of the movie duration, then merging is impossible; another merge target should be selected; otherwise, stream S will extend its lifetime to the end of the movie. According to this rule, when the interval size of multicasting exceeds half the length of a movie, fewer and fewer streams are able to find merge targets. It is worse in BEP's case in that patching streams depend heavily on periodical multicasting streams. Consequently, more streams have an extended lifetime until the end of the movie. These streams are non-periodical multicasting streams. They become candidate merge targets for patching streams and hence may help with performance. The reason performance fluctuations happen only to BEP rather than M. ERMT with MC is because unlike

BEP, M. ERMT in nature does not depend on periodical multicasting streams, which is essential to BEP due to its patching technique characteristic. Moreover, BEP is only able to build a merge tree with a maximum of two layers. This fact makes BEP more sensitive to unevenly generated client requests from time to time. As a result, the performance of BEP fluctuates.

- It hurts M. ERMT's performance when client buffer sizes are too big, as patching streams are able to merge into earlier targets. Hence, patching processes last too long. M. ERMT with MC provides multicasting streams that can be used as merge targets. Consequently, shorter patching periods become possible. That is why M. ERMT with MC slightly outperforms M. ERMT and M. ERMT without client buffer VCR support when w and d are greater than a certain value.

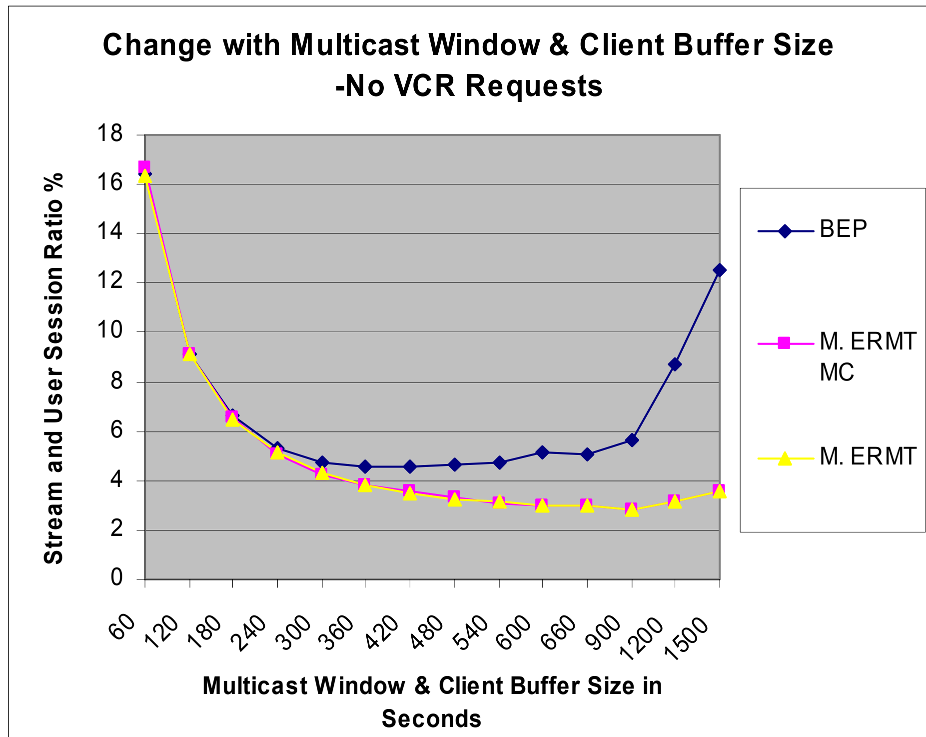

Figure 13 Performance without VCR Requests—Multicasting Interval and Client Buffer Size Change Together

- Input parameters of Figure 13: same as Figure 11's ($\lambda = 5$ per minute) except clients are not interactive; they always stay in play state until the end of the movie is reached.
- Observations and analyses of the result set: simulation for M. ERMT without client local buffer VCR support is absent since it is identical to M. ERMT when clients are not VCR interactive. For all three simulators illustrated in Figure 13, the performance is much better than the previous dataset that includes VCR operations. The ratios of all simulators drop rapidly from around 16.5% to 4.5% when w increases from 60 to 300 seconds (d = 60 to 300 seconds in the case of M. ERMT). They stay flat for BEP when w varies

from 300 to 900 seconds and go up rapidly after w ≥ 900 seconds. The ratios are flat for M. ERMT and M. ERMT with CM when d ranges from 360 to 1500 seconds. M. ERMT and M. ERMT with MC perform exactly the same in this figure because M. ERMT schemes do not depend on fixed-interval multicasting. Without VCR requests, the relationship among clients' play-points are fixed; thus, the multicasting streams generated by M. ERMT with MC are easily merged into targeting streams in a short period of time. As a result, the number of streams used to support the same amount of client requests are almost the same as M. ERMT's. They both outperformed BEP in the range where d is between 300 and1500 seconds. The performance gap becomes greater when d's value increases. BEP's worsening performance when multicasting interval size increases is due to the patching schemes' dependency on fixed-interval multicasting. As the fixed-interval multicasting window gets bigger, patching streams have to stay longer in order to merge into target streams. Therefore, it takes more stream time to support the same amount of client requests.

In the case of VCR operations, when λ is fixed and the client buffer size and fixed-interval size vary together, M. ERMT greatly outperforms BEP. M. ERMT also outperforms M. ERMT with MC until the client buffer size is set at greater than 1200 seconds. With the optimal client buffer setting, M. ERMT performs the best among all schemes. Client buffer sizes that are bigger than the optimal setting damage M. ERMT's performance as merging streams' lifetimes are extended. There exist the most favorable

multicasting interval sizes for BEP and M. ERMT with MC. When the sizes are small, too many multicasting streams are generated, which wastes system resources. In contrast, an oversized multicast window means that too few multicasting streams are created that are not able to accommodate all client requests. As a consequence, less resource-sharing takes place. Oversized multicasting intervals impact BEP more since it depends much more on fixed-interval multicasting compared to M. ERMT with MC. Without local buffer supporting VCR operations, M. ERMT's performance is definitely damaged by around 2% to 20% with a 10% average. Without VCR operations, M. ERMT and M. ERMT with MC perform the same. They outperform BEP more and more when d increases.

5.2.3.2 Impact of Multicasting Interval Size

In the previous section, the input workload parameters multicasting fixed-interval and client buffer size change together with same value in the simulations of BEP and M. ERMT with MC. The purpose of this section is to report on the performance change of BEP and M. ERMT with MC when only the periodical multicast's fixed-interval size w varies. The performance results are then compared with those of M. ERMT and M. ERMT without client local buffer VCR support, which have the same input parameters except that M. ERMT and M. ERMT without client local buffer VCR support do not use w at all (as they do not require fixed-interval multicasting). Client buffer size d is 2700 seconds for all the data points shown in Figure 14 below, while only the multicasting interval size w changes from 38 seconds to 2700 seconds. Hence, M. ERMT's line in Figure 14 is perfectly flat. So is the line of M. ERMT without client local buffer VCR

support. Simulations have also been run with d at 2400 seconds where w varies from 60 seconds to 2400 seconds. Similar simulations are performed when d is set to 2100, 1800, etc., until 300 seconds. Based on these simulation results, two three-dimensional figures, Figure 15 and Figure 16, are created respectively for BEP and M. ERMT with MC. Together, these figures display the corresponding performance changes. Figure 17 is a portion of Figure 12. It shows how M. ERMT and M. ERMT without client buffer VCR support perform with different client buffer sizes. Here, Figure 17 is for comparison purposes. The simulation results in Figure 17 are compared to those of Figure 15 and 16.

For all the data sets shown from Figure 14 through Figure 17, following are the common fixed input workload parameters: interarrival time of client request is 12 seconds; corresponding interarrival rate λ (requests per minute) is 5. Clients are VCR-interactive.

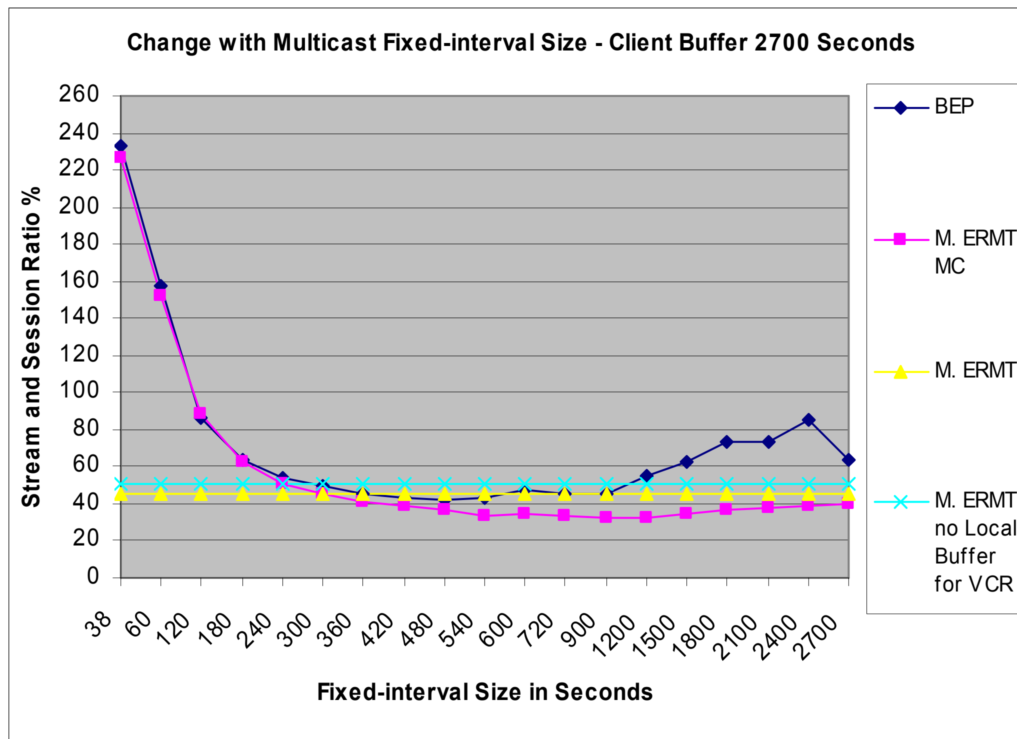

Figure 14 Performance with VCR—d = 2700 Seconds, w Changes

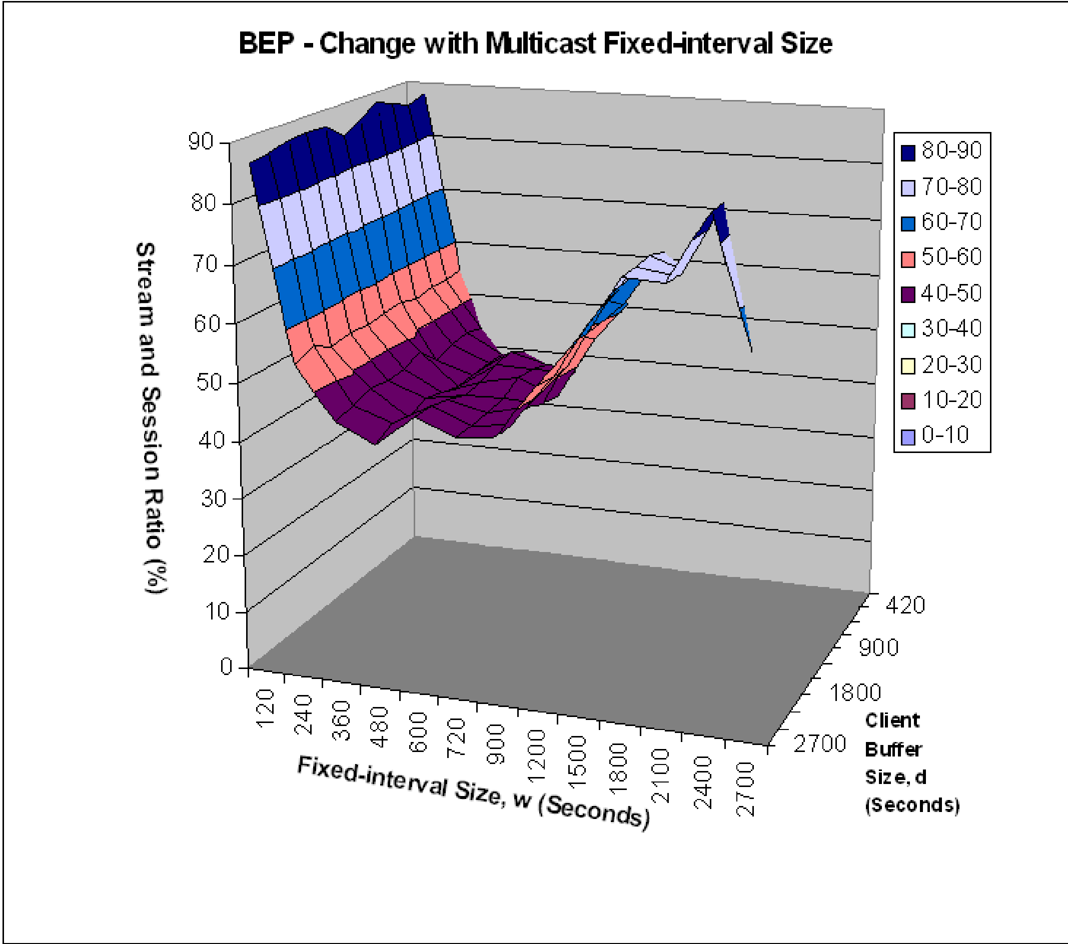

Figure 15 BEP Performance with VCR

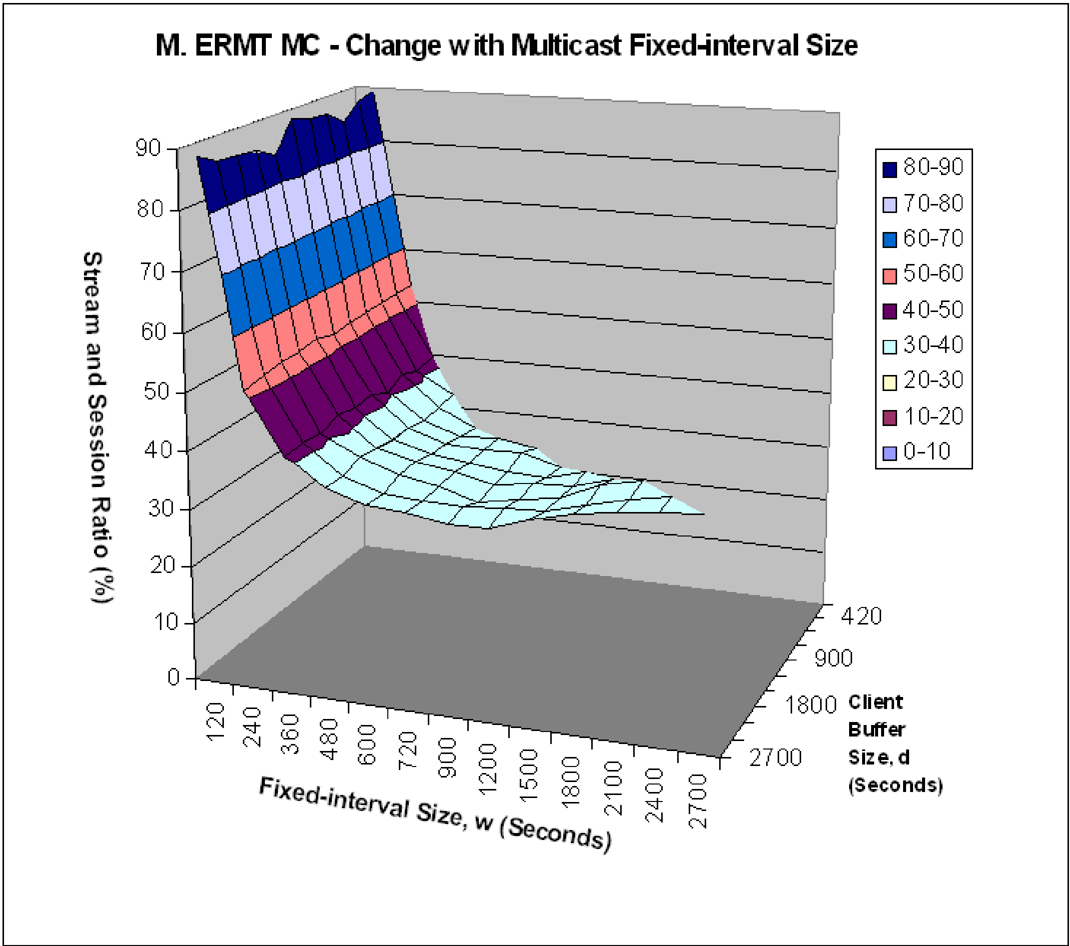

Figure 16 M. ERMT MC Performance with VCR

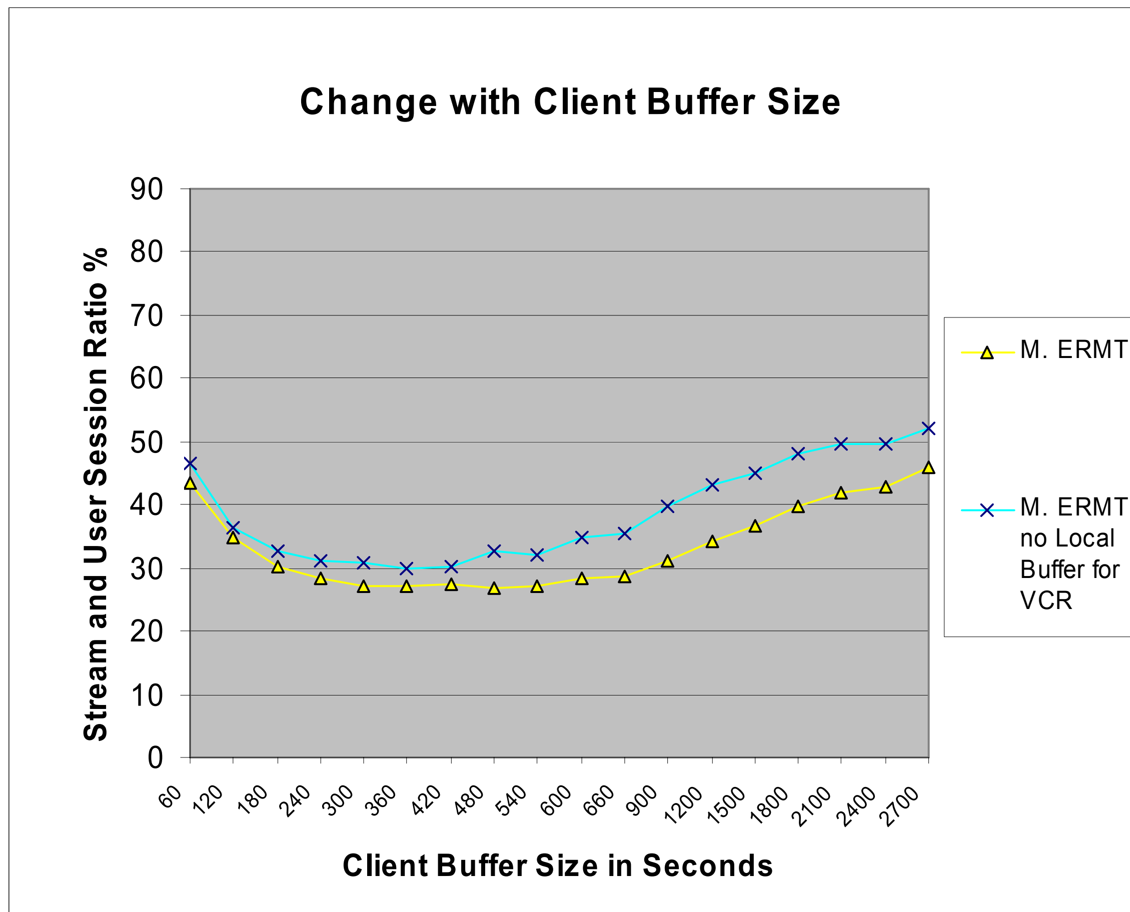

Figure 17 Performance with VCR—d Changes

5.2.3.2.1 Observations

a. The changing input parameter within each simulation is w, the interval size of fixed-interval multicasting streams. It varies from 38, 60, or 120 seconds to the value of client buffer size d.

b. The changing input parameter among all the simulations is client buffer size d. It varies from 300 to 2700 seconds from simulation to simulation.

c. In Figure 15 and 16 when d is less than or equal to 900 seconds, both BEP and M. ERMT with MC's result ratios decrease first and then increase when w increases.
d. M. ERMT with MC always outperforms BEP in comparing Figures 15 and 16. The performance gaps are enlarged when multicasting interval size w increases.
e. For given input parameters, Figures 15 and 16 (with such w values available) show that the same fixed-interval size w results in similar performance in the simulations even though the values of d are different. The figures illustrate that w ranging from 480 to 600 seconds offers the best performance ratios (around 43%) for BEP and that w ranging from 720 to 900 seconds provides the best ratios (around 32%) for M. ERMT with MC, regardless of client buffer size d.
f. M. ERMT outperforms BEP and M. ERMT with MC when client buffer size d ranges between 300 and 1200 seconds. M. ERMT performs the best (with a ratio of 26.7%) when d is set to 480 seconds. That is the best performance among all simulations with the same given particular input parameter settings described in this section.
g. When d ranges from 1500 through 2700 seconds, some of the result ratios of M. ERMT with MC are lower than M. ERMT's. M. ERMT performs better than M. ERMT without client buffer VCR support in all the simulations.

5.2.3.2.2 Analysis

According to the above observations, a M. ERMT scheme with client buffer size d set to 480 seconds seems the best choice in this particular scenario given the input workload parameter settings of the simulations. The simulation results again indicate that bigger

client local buffer sizes do not always benefit. Multicasting does help M. ERMT with MC's performance when the client local buffer size increases. The reasons behind these facts are related to the nature of M. ERMT. When the client local buffer allows, EMRT intends to build the biggest merge tree possible. Merge trees get deeper and bigger as client local buffer size increases. As a result, many long-term merges take place and create many "aged" streams, which increase the result ratio and therefore impact the performance. By introducing multicasting streams, smaller merge trees are formed instead and in turn improve the overall performance. The bigger client buffer benefits VCR support for any resource-sharing techniques, including M. ERMT. Yet it harms M. ERMT's performance due to its effect on the merge tree size. It will be better if the best merge tree size can be calculated with a given interarrival time and VCR interactivity rate. In that case, M. ERMT will be able to both maximize its strength in stream merging and fully utilize big client local buffers at the same time. M. ERMT with MC outperforms BEP while multicasting interval size increases since M. ERMT with MC depends much less on fixed-interval multicasting.

5.2.3.2 Verification of Suggested Interval Size of Periodical Multicasting

Section 4.2 mentioned a formula, which the authors of paper [26] claim can be used to calculate the optimal fixed interval size for periodical stream multicasting. Based on the interarrival rate given in 5.2.3.2 and the formula, the best interval size is 38 seconds after calculation. Here is the calculation:

Formula for Calculating Fixed Multicasting Interval Size

$$x = 2T / \sqrt{2M} \qquad [26]$$

x: the interval size for a video in terms of time

T and M: over a period of time T, there are M requests for the video on average.

$T = 60$ seconds and $M = 5$ according to the input workload parameter settings of Section 5.2.3.2. Thus $x = 38$ seconds after calculation.

This calculation result is used in the simulations. Figure 14 shows that both BEP and M. ERMT with MC perform the worst with this multicasting interval size. Therefore, the formula failed at the verification.

6. Conclusion and Future Works

Resource-sharing techniques are widely used by VOD servers. Stream merging is one of the most efficient resource-sharing techniques. ERMT is able to achieve merge trees with the closest cost of optimal merge tree. Full VCR support has become a “must have” feature for VOD services. This researcher proposed an algorithm to enable VCR support on ERMT. Furthermore, client local buffer and fixed-interval periodical multicasting were also deployed by the algorithm to improve the stream-client ratio. After thorough runs of simulations and numerous comparisons to BEP, the highly efficient resource-sharing technique, the proposed algorithm with client local buffer utilization and fixed-interval multicasting showed better performance in all simulations. The biggest discovery is that the best-performer is modified ERMT with client local buffer support for VCR without fixed-interval multicasting. Another discovery is that bigger client buffer size hurts the performance of ERMT. Future work may proceed in several directions. The first direction is to find a method that is able to provide the best client local buffer size or merge tree size for ERMT. Second, another method should be developed to determine the

optimal interval size of periodical multicasting. These methods must take both popularity of the video content and the degree of client interactivity into consideration. Third, peer-to-peer VCR support should be added and examined. Fourth, the performance of the proposed algorithm must be evaluated in a clustered server environment.

References:

[1] Raj Jain, *the Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Digital Equipment Corporation, Littleton, Massachusetts, 1991

[2] Suneuy Kim, Chita R. Das, "An Analytical Model for Interval Caching in Interactive Video Servers", *Journal of Network and Computer Applications* 30 (2007) pp. 384-413

[3] A. Dan, et al., "Scheduling Policies for an On-Demand Video Server with Batching" *Proc. ACM Multimedia*, 1994

[4] C. C. Aggarwal, J. Wolf, P. S. Yu, "On Optimal Piggybacking Merging Policies for Video-on-Demand Systems," *ACM Sigmetrics*, vol. 24, May 1996, pp. 200-209

[5] K. A. Hua, Y. Cai, and S. Sheu, "Patching: A Multicast Technique for True Video-on-Demand Services," *ACM Multimedia'98*, Bristol, U.K., Sep. 1998, pp. 191-200

[6] R. Tewari, H. Vin, A. Dan, D. Sitaram, "Resource-Based Caching for Web Servers", *Proc. of SPIE/ACM Conference on Multimedia Computing and Networking*, San Jose, CA, Jan. 1998, pp. 191-204

[7] Almeida, J. M., D. L. Eager, and M. K. Vernon, “A Hybrid Caching Strategy for Streaming Media Files”, *Proc. Multimedia Computing and Networking 2001*, San Jose, CA, Jan. 2001

[8] A. Dan, et al., “A Generalized Interval Caching Policy for Mixed Interactive and Long Video Workloads” *Proc. Multimedia Computing and Networking*, 1996

[9] D. Eager, M. Vernon, “Dynamic Skyscraper Broadcasts for Video-on-Demand”. *The Fourth International Workshop on Multimedia Information Systems*, Istanbul, Turkey, Sep. 1998

[10] D. Eager, M. Vernon, J. Zahorjan, “Minimizing Bandwidth Requirements for On-Demand Data Delivery”, *IEEE Transactions on Knowledge and Data Engineering* 13, Sep. 2001, pp.742-757

[11] A. Bar-Noy, R.E. Ladner, “Competitive on-line Stream Merging Algorithm for Media-on-Demand”, *Proc. SODA’01*, 2001

[12] E. G. Coffman Jr., P. Jelenkovic, P. Momcilovic, “The Dyadic Algorithm for Stream Merging”, *Journal of Algorithms*, 43(1), 2002, pp.120-137

[13] H. Ma, G. Shin, W. Wu, “Best-Effort Patching for Multicast True VoD Service”, *ACM Multimedia Tools and Applications*, Vol. 26, Issue 1, May 2005. pp. 101 - 122

[14] Nelson L. S. da Fonseca, Hana Karina S. Rubinsztejn, “Dimensioning the Capacity of True Video-on-Demand Servers”, *Multimedia, IEEE Transactions*, Vol. 7, Issue 5, Oct. 2005, pp. 932-941

[15] E. L. Abram-Profeta, K.G. Shin, “Providing Unrestricted VCD Functions in Multicast Video-on-Demand Servers”, *Proc. Multimedia Computing and Systems, IEEE International Conference, 28 June – 1 July* 1998, pp. 66-75

[16] W. F. Poon, K. T. Lo, J. Feng, “Providing VCR Functionality in Multicast Video-on-Demand Systems Using Adaptive Batching”, *Journal of Visual Communication and Image Representation*, vol. 13, issue 4, December 2002, pp. 476-489

[17] Nelson L. S. da Fonseca, Hana Karina S. Rubinsztejn, “Channel Allocation in True Video-on-Demand systems”, *Global Telecommunications Conference, 2001.* Vol. 3, 2001

[18] M. Y. Y. Leung, J. C. S. Lui, L. Golubchik, “Use of Analytical Performance Models for System Sizing and Resource Allocation in Interactive Video-on-Demand Systems Employing Data Sharing Techniques”, *Knowledge and Data Engineering, IEEE Transactions, vol. 14, issue: 3, May ~ June 2002,* pp. 615-637

[19] S.W. Carter, D. D.E. Long, J-F Paris, “An Efficient Implementation of Interactive Video-on-Demand”, *Eighth IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'00)* pp.172

[20] K. A. Hua, S. Sheu, J. Z. Wang, “Earthworm: A Network Memory Management Technique for Large-scale Distributed Multimedia Applications”, in *Proc. IEEE INFOCOM /97*, Vol.3, Kobe, Japan, Apr. 1997, pp.990-997

[21] S. Shen, K. A. Hua, W. Tavanapong, “Chaining: A Generalized Batching Technique for Video-On-Demand”, *1997 International Conference on Multimedia Computing and Systems (ICMCS'97),* pp.110

[22] T-C. Su, S-Y. Huang, C-L. Chan, J-S.Wang, “Optimal Chaining Scheme for Video-on-Demand Applications on Collaborative Networks”, *IEEE Transactions on Multimedia*, Vol.7, No.5, October 2005

[23] D. Eager, M. Vernon, J. Zahorjan, “Bandwidth Skimming: A Technique for Cost-Effective Video-on-Demand”, *Proc. Multimedia Computing and Networking 2000*, San Jose, CA January 2000

[24] M. Rocha, M. Maia, Í. Cunha, J. Almeida, S. Campos, “Scalable Media Streaming to Interactive Users”, *Proc. of the 13th annual ACM international conference on Multimedia* MULTIMEDIA '05 , November 6-11, 2005

[25] J.J Schultz, T. Znati, “An efficient scheme for chaining with client-centric buffer reservation for multi-media streaming”, *Simulation Symposium, 2003. 36th Annual* 30 Mar. - 2 Apr. 2003, pp. 31-38

[26] S. Jin, A. Bestavros, “Scalability of Multicast Delivery for Non-sequential Streaming Access”, *ACM SIGMETRICS Performance Evaluation Review, Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* SIGMETRICS '02, Vol.30, Issue 1

[27] R.T. Shi, L. Shao, Y.Z. Pei, D. Xie, “A Novel Stream Merging Algorithm for VOD Servers”, *Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. Proc. the 2003 Joint Conference of the Fourth International Conference,* Vol. 3, 15-18 Dec. 2003. pp. 1982 – 1986

[28] H. Ma, K. G. Shin, “Multicast Video-on-Demand Services”, *ACM SIGCOMM Computer Communication Review*, Vol. 32, Issue 1 , January 2002

[29] J-F. Paris, “A Cooperative Distribution Protocol for Video-on-Demand”, *Proc. Sixth Mexican International Conference on Computer Science (ENC'05)*, September 2005, pp. 240-247

[30] C. P. Costa, I. S. Cunha, A. Borges, C. V. Ramos, M. M. Rocha, J. M. Almeida, B.Ribeiro-Neto, “Workload Analysis: Analyzing Client Interactivity In Streaming Media”, *Proc. of the 13th international conference on World Wide Web*, WWW '04, May 2004

[31] D. Eager, M. Vernon, J. Zahorjan, “Optimal and Efficient Merging Schedules for Video-on-Demand Servers”, *Proc. of the 17th ACM International Conference on Multimedia (Part 1)*, October 30 - November 05, 1999, Orlando, Florida, United States. pp.199 – 202

[32] A. Bar-Noy, J. Goshi, R. Ladner, K. Tam, “Comparison of Stream Merging Algorithms for Media-on-demand”, *Multimedia Systems, Springer-Verlag 2004,* 9: 411 – 424 (2004)

[33] Products/FAQ : “Introduction to CSIM 19”, *Mesquite Software Inc. 2005*, http://www.mesquite.com/products/csimprim.htm

Appendix A: Simulation Results

Each simulation = 10 hours of simtime * 100 runs
Number of movies = 1
Movie Length = 5400 seconds

Interactive				**BEP**			**ERMT with MC**			**ERMT**			**ERMT w/o Client Buffer for VCR**		
Λ(request/minute)	Interarrival time(seconds)	Multicast w/ Fixed-Interval w (seconds)	Client Buffer Size d (seconds)	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**
1	**60**	300	300	15.98	9.52	**187.2**	16.05	9.39	**198.3**	5.07	9.07	**57.8**	5.47	9.69	**59.38**
2	**30**	300	300	18.66	19.6	**99.39**	18.68	19.4	**99.52**	8.38	20.15	**42.47**	8.53	19.92	**43.82**
3	**20**	300	300	20.58	29.81	**70.91**	20.19	30.6	**67.52**	10.18	29.62	**35**	11.46	30.63	**37.69**
4	**15**	300	300	22.83	40.85	**56.42**	21.45	39.3	**55.82**	12.36	40.26	**30.81**	12.86	39.82	**32.39**
5	**12**	300	300	24.32	50.04	**48.74**	23.01	50.8	**45.72**	13.68	50.23	**26.98**	14.49	50.52	**28.62**
6	**10**	300	300	26.05	60.22	**43.28**	23.67	59.9	**39.79**	14.43	59.89	**23.9**	16.64	61.31	**27.1**
7	**8.6**	300	300	27.6	70.99	**38.79**	24.62	71.2	**34.4**	16.23	72	**22.24**	18.32	72	**25.1**
8	**7.5**	300	300	29.92	82.29	**36.14**	25.5	82.3	**30.84**	16.61	81.37	**20.05**	19.18	82.5	**22.94**

9	**6.7**	300	300	30.88	91.1	**33.7**	26.23	91.1	**28.59**	18.26	92.94	**19.28**	20.33	90.56	**22.14**
10	**6**	300	300	34.04	101.5	**33.18**	27.22	103	**26.27**	19.19	101.3	**18.52**	21.84	101.6	**21.15**
No VCR				**BEP**			**ERMT with MC**			**ERMT**			**ERMT w/o Client Buffer for VCR**		
Λ(request/minute)	Interarrival time(seconds)	Multicast w/ Fixed-Interval w (seconds)	Client Buffer Size d (seconds)	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**
1	**60**	300	300	15.77	88.63	**17.51**	15.78	87	**17.84**	15.75	87.29	**17.74**			
2	**30**	300	300	18.38	175	**10.07**	18.19	175	**9.9**	18.07	172.4	**10.03**			
3	**20**	300	300	20.19	261.6	**7.22**	19.38	263	**6.89**	19.42	262.5	**6.94**			
4	**15**	300	300	21.92	349.2	**5.79**	20.22	348	**5.31**	20.4	350.4	**5.31**			
5	**12**	300	300	22.88	442.8	**4.73**	21.06	436	**4.26**	21.11	437.4	**4.3**	**Same as ERMT**		
6	**10**	300	300	24.85	523.3	**4.22**	21.71	527	**3.62**	21.86	523.1	**3.73**			
7	**8.6**	300	300	27.12	608.7	**3.99**	22.24	605	**3.11**	22.42	611.4	**3.1**			
8	**7.5**	300	300	28.01	700.1	**3.54**	23.09	704	**2.88**	22.66	695.1	**2.91**			
9	**6.7**	300	300	29.61	791.9	**3.27**	23.55	792	**2.45**	23.52	789.4	**2.4**			
10	**6**	300	300												
Interactive				**BEP**			**ERMT with MC**			**ERMT**			**ERMT w/o Client Buffer for VCR**		
Λ(request/minute)	Interarrival time(seconds)	Multicast w/ Fixed-Interval w (seconds)	Client Buffer Size d (seconds)	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**	Current No. of Streams	Current No. of Sessions	Channl/Session **Ratio(%)**
5	12	**60**	**300**	77.58	52.15	**150.8**	77.73	51.8	**152.7**	22.95	52.97	**43.53**	23.81	51.33	**46.47**
5	12	**120**	**300**	44.68	51.56	**87.78**	44.69	51.7	**87.79**	17.83	51.14	**34.96**	18.97	52.09	**36.31**
5	12	**180**	**300**	33.08	51.78	**64.45**	32.34	51.4	**63.7**	15.56	51.56	**30.14**	16.83	51.16	**32.82**
5	12	**240**	**300**	27.65	51.66	**53.76**	26.8	50.8	**53.01**	14.71	51.54	**28.35**	16.19	52.05	**31.05**
5	12	**300**	**300**	25.24	51.13	**49.51**	23.21	51.7	**45.26**	14.1	51.6	**27.22**	15.67	50.37	**30.91**
5	12	**360**	**360**	23.82	52.96	**45.14**	21.1	51.5	**41.27**	14.19	51.74	**27.26**	15.61	51.91	**29.84**
5	12	**420**	**420**	22.82	52.57	**43.29**	19.91	51.2	**39.21**	14.32	52.01	**27.46**	15.8	52.28	**30.23**
5	12	**480**	**480**	22.47	52.01	**42.91**	18.88	51.4	**36.89**	14.02	52.01	**26.7**	16.76	51.32	**32.52**
5	12	**540**	**540**	21.78	51.45	**42.3**	18.91	51.2	**36.96**	13.93	51.18	**27.17**	16.79	52.16	**32.1**
5	12	**600**	**600**	23.88	51.87	**45.89**	17.89	51.3	**34.96**	14.85	51.91	**28.43**	17.71	50.95	**34.8**
5	12	**660**	**660**	21.83	51.9	**42.03**	17.76	50.3	**35.44**	14.69	51.23	**28.64**	18.17	51.29	**35.37**
5	12	**900**	**900**	24.15	50.85	**47.45**	17.33	51.7	**33.61**	16.12	51.56	**31.14**	20.2	50.93	**39.7**
5	12	**1200**	**1200**	28.81	51.01	**56.4**	17.38	51.1	**34.12**	17.92	52.52	**34.26**	22.32	51.8	**43.16**
5	12	**1500**	**1500**	33.22	51.78	**63.82**	17.84	52.9	**33.75**	18.66	51.3	**36.68**	23.15	51.67	**44.96**
5	12	**1800**	**1800**	37.87	51.94	**72.52**	18.94	52.2	**36.24**	20.63	52.43	**39.69**	24.5	51.12	**48.15**
5	12	**2100**	**2100**	37.82	51.51	**72.99**	19.93	51.6	**38.5**	21.66	51.76	**41.99**	25.81	52.09	**49.67**
5	12	**2400**	**2400**	44.79	52.5	**84.96**	20.56	52.7	**39.1**	22.04	51.56	**42.78**	25.87	52.36	**49.58**
5	12	**2700**	**2700**	33.07	52.32	**63.18**	20.62	50.6	**40.95**	23.64	51.61	**46.05**	26.68	51.42	**52.09**
5	12	**3000**	**3000**	38.59	51.61	**74.4**	20.66	51	**40.64**	22.84	51.46	**44.59**	26.68	51.42	**52.09**
5	12	**3500**	**3500**	28.67	51.48	**55.21**	21.51	52	**41.6**	22.83	50.41	**45.62**	26.68	51.42	**52.09**
5	12	**4000**	**4000**	35.2	51.1	**68.63**	24.79	52.7	**47.4**	22.28	51.03	**43.98**	26.68	51.42	**52.09**
5	12	**4500**	**4500**	39.94	51.56	**77.28**	25.26	50.5	**50.73**	22.51	51.9	**43.87**	26.68	51.42	**52.09**
5	12	**5000**	**5000**	31.83	50.91	**62.15**	20.85	50.2	**41.94**	22.55	50.79	**44.61**	26.68	51.42	**52.09**
5	12	**5400**	**5400**	31.91	50.27	**63.1**	23.8	51.6	**46.6**	22.46	49.8	**45.26**	26.68	51.42	**52.09**

No VCR				BEP			ERMT with MC			ERMT			ERMT w/o Client Buffer for VCR		
Λ(request/minute)	Interarrival time(seconds)	Multicast w/ Fixed-Interval w (seconds)	Client Buffer Size d (seconds)	Current No. of Streams	Current No. of Sessions	**Channel/Session Ratio(%)**	Current No. of Streams	Current No. of Sessions	**Channel/Session Ratio(%)**	Current No. of Streams	Current No. of Sessions	**Channel/Session Ratio(%)**	Current No. of Streams	Current No. of Sessions	**Channel/Session Ratio(%)**
5	12	**60**	**300**	76.35	451.2	**16.45**	76.4	446	**16.69**	75.37	449.9	**16.31**			
5	12	**120**	**300**	43.09	447.1	**9.16**	42.79	444	**9.11**	42.69	443.5	**9.15**			
5	12	**180**	**300**	31.48	439.9	**6.66**	30.82	438	**6.52**	30.52	440.1	**6.43**			
5	12	**240**	**300**	25.57	441.3	**5.29**	24.62	439	**5.08**	24.43	437.5	**5.11**			
5	12	**300**	**300**	22.88	442.8	**4.73**	21.06	436	**4.26**	21.11	437.4	**4.3**			
5	12	**360**	**360**	22.02	437	**4.56**	18.68	434	**3.84**	18.57	431.1	**3.85**	**Same as ERMT**		
5	12	**420**	**420**	21.84	429.4	**4.59**	17.35	431	**3.55**	17.57	441	**3.48**			
5	12	**480**	**480**	21.52	422.1	**4.64**	16.09	426	**3.29**	15.93	419.6	**3.27**			
5	12	**540**	**540**	22.27	425.8	**4.75**	15.31	427	**3.05**	15.31	424.2	**3.16**			
5	12	**600**	**600**	23.64	422.4	**5.16**	14.4	415	**3.01**	14.69	424.1	**3.01**			
5	12	**660**	**660**	23.67	428.3	**5.04**	14.57	426	**2.97**	14.47	421.8	**2.95**			
5	12	**900**	**900**	25.33	413.5	**5.65**	13.99	417	**2.85**	13.73	412.2	**2.84**			
5	12	**1200**	**1200**	34.65	377.4	**8.7**	13.72	378	**3.14**	13.63	375.3	**3.13**			
5	12	**1500**	**1500**	46.85	358.5	**12.56**	14.29	357	**3.53**	14.49	359	**3.53**			

Each simulation = 10 hr * 100 runs

Number of movies = 1

Movie Length = 5400 seconds

Interarrival Time = 12 Seconds

Λ(request/minute) = 5

Interactive = True

		BEP			**ERMT with MC**			**ERMT**	
Multicast w/ Fixed-Interval w (seconds)	Client Buffer Size d (seconds)	total Streams	total Sessions	**Ratio(%)**	total Streams	total Sessions	**Ratio(%)**	**ERMT Ratio(%)**	**ERMT w/o Client Buffer VCR Support Ratio(%)**
38	**2700**	109.55	49.92	**233.26**	109.75	49.53	**226.14**	**46.05**	**52.09**
60	**2700**	77.06	50.01	**157.02**	76.93	51.44	**152.24**	**46.05**	**52.09**
120	**2700**	44.33	51.59	**86.83**	43.94	50.08	**88.99**	**46.05**	**52.09**
180	**2700**	32.43	51.05	**64.12**	32	52.02	**62.61**	**46.05**	**52.09**
240	**2700**	26.91	50.15	**54.17**	26.02	51.4	**51.19**	**46.05**	**52.09**
300	**2700**	25.04	50.83	**49.52**	22.77	51.09	**45.03**	**46.05**	**52.09**
360	**2700**	23.31	51.5	**45.36**	21.21	51.85	**40.99**	**46.05**	**52.09**
420	**2700**	21.93	50.98	**42.89**	19.58	50.92	**38.77**	**46.05**	**52.09**
480	**2700**	21.32	50.55	**42.31**	18.36	50.2	**36.59**	**46.05**	**52.09**
540	**2700**	21.85	50.39	**43.17**	17.72	52.84	**33.48**	**46.05**	**52.09**
600	**2700**	23.77	49.77	**47.64**	17.34	50.59	**34.31**	**46.05**	**52.09**
720	**2700**	22.99	51.06	**44.89**	17.5	51.52	**33.93**	**46.05**	**52.09**
900	**2700**	23.92	51.95	**45.66**	16.91	51.45	**32.81**	**46.05**	**52.09**
1200	**2700**	27.25	49.24	**54.87**	16.52	50.27	**32.78**	**46.05**	**52.09**
1500	**2700**	32.41	51.03	**62.94**	17.49	49.9	**35.03**	**46.05**	**52.09**

1800	2700	36.53	49.86	73.3	18.64	50.37	37.11	46.05	52.09
2100	2700	37.56	51.31	73.17	19.43	51.33	38.05	46.05	52.09
2400	2700	42.55	50.03	84.73	19.47	50.21	38.89	46.05	52.09
2700	2700	32.34	51.05	63.12	19.97	51.1	39.46	46.05	52.09

120	2400	43.98	50.84	87.45	44.17	51.19	87.28	42.78	49.58
240	2400	27.39	49.63	55.74	26.07	50.59	52.04	42.78	49.58
360	2400	22.9	49.69	46.17	21.19	51.71	41.21	42.78	49.58
480	2400	22.12	50.45	43.74	18.49	51.05	36.29	42.78	49.58
600	2400	23.89	50.18	47.49	17.17	50.6	34.14	42.78	49.58
720	2400	22.31	50.17	44.69	17.18	50.82	33.82	42.78	49.58
900	2400	23.66	51.42	45.62	16.71	49.99	33.38	42.78	49.58
1200	2400	27.96	51.02	54.42	16.99	49.74	34.13	42.78	49.58
1500	2400	33.04	50.87	64.61	17.43	50.88	34.41	42.78	49.58
1800	2400	37.42	50.69	73.55	18.91	52.43	36.18	42.78	49.58
2100	2400	36.83	50.66	72.35	19.15	50.47	37.92	42.78	49.58
2400	2400	43.99	51.7	84.61	19.98	50.96	39.21	42.78	49.58

120	2100	44.15	50.54	88.4	44.01	51.33	86.88	41.99	49.67
240	2100	26.99	50.62	53.72	26.27	50.41	52.54	41.99	49.67
360	2100	23.55	50.17	47.17	20.85	51.6	40.66	41.99	49.67
480	2100	21.65	50.46	42.88	18.46	49.52	37.52	41.99	49.67
600	2100	23.42	50.06	46.67	17.56	50.28	35.07	41.99	49.67
720	2100	23.51	52.11	44.99	17.22	50.74	33.98	41.99	49.67
900	2100	23.66	50.4	46.66	16.36	51.6	31.89	41.99	49.67
1200	2100	27.97	51.41	54.27	16.81	50.94	33.29	41.99	49.67
1500	2100	33.36	50.42	65.76	17.54	50.96	34.43	41.99	49.67
1800	2100	38.2	51.32	73.91	19.01	51.1	37.36	41.99	49.67
2100	2100	36.68	51.19	71.16	19.57	50.39	38.86	41.99	49.67

120	1800	44.02	50.42	88.5	44.08	51.81	86.39	39.69	48.15
240	1800	27.84	51.48	54.53	26.35	50.5	52.58	39.69	48.15
360	1800	23.22	50.43	46.39	20.91	50.13	42.02	39.69	48.15
480	1800	22.3	50.73	44.25	18.3	49.86	36.84	39.69	48.15
600	1800	22.69	49.87	45.22	17.55	50.05	35.26	39.69	48.15
720	1800	23.25	50.94	45.47	17.29	50.36	34.47	39.69	48.15
900	1800	23.27	50.89	45.41	16.69	51.11	32.83	39.69	48.15
1200	1800	29.57	51.17	57.44	17.48	51.72	33.86	39.69	48.15
1500	1800	32.62	51.21	63.52	17.42	50.68	34.32	39.69	48.15
1800	1800	36.73	50.31	72.79	18.08	50.12	36.17	39.69	48.15

120	1500	44.07	50.64	88.19	43.92	52.5	84.52	36.68	44.96
240	1500	27.38	51.1	53.87	26.26	50.02	53.16	36.68	44.96
360	1500	23.45	50.31	46.84	20.58	50.38	41.22	36.68	44.96
480	1500	21.42	49.76	43.17	18.48	50.88	36.41	36.68	44.96
600	1500	23.17	50.61	45.68	17.54	51.06	34.48	36.68	44.96
720	1500	23.44	49.51	47.29	17.35	51.49	33.71	36.68	44.96

900	1500	22.52	50.14	44.63	16.9	51.26	33.03	36.68	44.96
1200	1500	27.61	49.54	55.38	16.62	51.16	32.62	36.68	44.96
1500	1500	32.64	51.11	63.54	17.83	51.51	34.64	36.68	44.96

120	1200	43.92	51.66	85.73	44.25	49.92	89.71	34.26	43.16
240	1200	27.02	50.25	54.3	26.37	51.2	51.97	34.26	43.16
360	1200	23.75	49.81	47.93	20.24	48.87	41.71	34.26	43.16
480	1200	21.36	50.32	42.53	18.27	50.04	36.56	34.26	43.16
600	1200	24.21	51.32	47.07	17.11	49.83	34.42	34.26	43.16
720	1200	24.4	51.55	46.94	17.45	51.48	33.91	34.26	43.16
900	1200	24.74	52.12	47.15	16.78	49.23	34.29	34.26	43.16
1200	1200	27.96	51.13	54.45	16.73	51.43	32.44	34.26	43.16

120	900	44.37	51.31	87.5	43.81	50.31	88.63	31.14	39.7
240	900	27.27	50.21	54.83	26.32	50.26	52.49	31.14	39.7
360	900	23.58	52.42	44.86	20.62	51.39	40.47	31.14	39.7
480	900	21.44	49.99	42.84	18.3	51.49	35.74	31.14	39.7
600	900	24.41	51.12	47.77	17.14	51.23	33.58	31.14	39.7
720	900	23.34	50.58	46.14	16.98	50.05	33.94	31.14	39.7
900	900	22.95	50.16	45.56	16.14	48.75	33.19	31.14	39.7

120	660	44.14	50.12	89.33	43.77	50.29	88.15	28.64	35.37
240	660	27.56	51.85	53.64	26.24	50.52	52.59	28.64	35.37
360	660	23.32	50.86	45.99	20.77	49.93	42.02	28.64	35.37
480	660	21.9	50.83	43.17	18.42	50.84	36.37	28.64	35.37
600	660	23.76	51.55	45.88	17.07	51.32	33.32	28.64	35.37
660	660	20.67	49.86	41.27	17.21	51.44	33.53	28.64	35.37

120	540	44.13	51.08	88.09	43.86	51.51	85.91	27.17	32.1
240	540	27.23	50.37	54.33	26.23	50.19	52.86	27.17	32.1
360	540	23.81	51.35	46.44	20.59	51.32	40.49	27.17	32.1
480	540	22.01	51.35	42.71	18.44	50.55	36.74	27.17	32.1
540	540	21.09	50.31	42.03	17.55	50.63	34.8	27.17	32.1

120	420	44.03	51.24	86.85	43.95	50.62	88.46	27.46	30.23
240	420	27.69	51.56	54.15	26.26	50.32	52.95	27.46	30.23
360	420	23.29	51.27	45.37	20.73	50.37	41.64	27.46	30.23
420	420	21.75	50.66	43.12	19.4	50.41	38.66	27.46	30.23

60	300	77.44	50.91	154.5	77.46	51.38	152.97	27.22	30.91
120	300	44.17	50.85	87.98	44.15	49.96	89.48	27.22	30.91
180	300	32.86	51.42	64.7	32.27	52.38	62.29	27.22	30.91
240	300	27.41	50.41	54.65	25.99	49.63	53.05	27.22	30.91
300	300	24.32	50.04	48.74	23.01	50.77	45.72	27.22	30.91

Appendix B: Source Code

```
/*****************************************************************************************
***
 ERMT.h:
 Authors: Ngyat tsui
 Student ID: 002829568
 Project:  CS298 FALL 2008, SJSU
 Instructor: Dr.Suneuy Kim

*****************************************************************************************
**/
/* funtion definitions */
void init_movie(int num); //initializing movie array with durations.
void zipf_init();
int choose_movie();
void zipf_set();
void req_generator();
int Admission();
void result_print(int val);
void user_Session(struct Client * S);
void add_client(struct Client *C);
void remove_client(struct Client *C);
void release_one_client(struct Client* C);
void join_stream(struct Client *C, struct Stream *S);
void add_multicast_stream(struct Stream *S);
void insert_multicast_stream(struct Stream *S);
void remove_multicast_stream(struct Stream *S);
void release_multicast_stream(struct Stream *S);
void insert_non_multicast_stream(struct Stream* S);
void remove_non_multicast_stream(struct Stream *S);
void release_non_multicast_stream(struct Stream *S);
void release_kid_streams(struct Stream* S);
void stream_Session(struct Stream *S);
void add_results(int val, double *my_var);
void update_stream(struct Stream* S);
void merge_stream(struct Stream* S);
void release_clients(struct Stream* S);
void update_client_state(struct Client *C, int new_state, int new_offset, double
new_speed, int new_duration);
int on_local_buffer_only(struct Client *C);
int check_local_buffer(struct Client *C, double VCR_speed);
int in_buffer(struct Client *C, int playpoint);
void play(struct Client *C);
void play_from_jump(struct Client *C, int new_duration);
void stop(struct Client *C);
void pause(struct Client *C);
void jump(struct Client *C, int new_state);
void pic_VCR(struct Client *C, int new_state);
struct Stream * new_isolated_stream(struct Client *C);
int find_playpoint(struct Client *C);
void leave_current_stream(struct Client *C);
int local_buffer_full(struct Client *C);
void check_VCR(struct Client *C);
void maintain_VCR(struct Client *C);
int VCR_pick(int state);
double VCR_speed(int state);
int state_duration(int state);
int pick_jump_point(struct Client *C, int jump_direction);
struct Stream* Parent(struct Stream *S);
struct Stream* New_stream(int movieID, int offset);
struct Stream* Youngest_kid(struct Stream *Parent);
struct Client* New_client(struct Stream* S);
struct Stream* Better_parent (struct Stream *node, struct Stream *orphan);
struct Client* Last_client(struct Stream *S);
struct Stream*  New_fixed_interval_multicasting_stream(int movieID);
void update_time_last_kid_merges(struct Stream *S);


/*****************************************************************************************
**
 ERMT.c:
 Authors: Ngyat tsui
 Student ID: 002829568
 Project:  CS298 FALL 2008, SJSU
 Instructor: Dr.Suneuy Kim

*****************************************************************************************
**/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "csim.h"
#include "ERMT.h"

/*unit for all time period define below is: second*/
#define S_TIME 36000   //user: how long you want to run this program for one time. 9000
on the left means
                        //      that the simulation will stop when the playbacks of 9000
requests are over
#define N_RUNS 100        //user: how many runs you want for this program
#define MAX_NUM_MOVIES 1200 //don't change this, this is for programers only
#define NUM_MOVIES 1  //user: how many movies are there on the server; please don't
change it bigger than 1200;
                        //zipf needs this too
#define TRUE 1
#define FALSE 0
#define DEBUG 0 //Turn on Debug, set to 1; Turn off Debug, set to 0
#define FIXED_INTERVAL_MULTICAST 1 //Turn on fixed-interval multicast set to 1; Turn off,
set to 0

#define MAXUSERS 4000   //don't change this is for programmers only
#define DISKBW 160        //MBPS, fixed in this simulation, don't change
#define DISPLAYRATE 192       //KBPS MPEG1, fixed in this simulation, don't change
#define INTERVALDUR 1.0 //don't change this is for programmers only
#define SAFETY 20       //don't change this is for programmers only
#define IF_ONLY_ONE_MOVIE_LENGHT 90 //simplified for test in order to avoid hardcoding,
may not need it later

#define INTERARRIVAL_TIME 12.0 //user: the average interarrival time you want, unit =
seconds
#define IC_TRACKING_SIZE 800 //don't change this is for programmers only
#define CONF_LOWER 0.05 //user: the confidence interval you want
#define CONF_UPPER 0.95 //user: the confidence interval you want
#define VERY_INTERACTIVE 1 //VCR test mode: very interactive, set to 1; not very
interactive, set to 0;
                           //set to any other integer to turn off VCR interaction

#define CHANNELS 1000000           /*total channel capacity*/
#define MOVIES  500                     /*total number of movies*/
#define CLIENT_BUFFER   660     /*client buffer size in terms of play time*/
#define MULTICAST_INTERVAL  120 /*interval between multicast streams in terms of play
time*/
#define DURATION_FACTOR  1.0   /*duration factor used when return a VCR duration   0 <
DURATION_FACTOR <= 2.0*/
#define MULTICAST_DELAY  100   /*delay to close a finished multicast; so done clients can
terminate before the multicast does */

//stream type
#define MULTICAST 0
#define ADMISSION 1
#define INTERACTIVE 2
#define MERGE 3
#define FI_MC 4

/* define state */
#define PLAY 0   /*Playback/Resume*/
#define PAUSE 1    /*Pause*/
#define JUMP_F 2   /* FF without picture (Jump) */
#define JUMP_B 3   /* RW without picture (Jump) */
#define FF 4 /* FF with picture */
#define RW 5 /* RW with picture */
#define SM 6 /* Slow-Motion with picture */
#define STOP 7 /*Stop = done playing*/

/* define state duration */
#define PLAY_Dura 600  /*Playback/Resume*/
#define PAUSE_Dura 300 /*Pause*/
#define JUMP_Dura 1  /* FF/RW without picture (Jump) */
#define FFRW_Dura 150/* FF/RW with picture */
#define SM_Dura 120  /*Slow-Motion with picture*/
#define STOP_Dura 1  /*Stop = done playing*/

/* define state speed */
#define PLAY_SPEED 1  /*regular play speed*/
#define IDLE_SPEED 0  /*speed at pause or stop*/

#define SLOW_MOTION_SPEED 0.5/* fixed slow motion speed*/
#define FF_SPEED 3 /* fixed FF speed*/
#define RW_SPEED -3 /* fixed RW speed*/
```

```
#define SM_F 0.5   /*Slow-Motion forward */
#define SM_B -0.5   /*Slow-Motion backward */
#define FF2 2 /*FF x2*/
#define FF4 4 /*FF x4*/
#define FF8 8 /*FF x8*/
#define RW1 -1 /*RW x1*/
#define RW2 -2 /*RW x2*/
#define RW4 -4 /*RW x4*/
#define RW8 -8 /*RW x8*/

typedef struct Client {
  int movieID;
  struct Stream *current_stream; /* pointer to a Stream */
  int state; /*current play state of the client */
  double speed; /*only for VCR+picture operations (FF, RW or Slow potion) */
  int offset; /*play point of previous update; need to change when 'state' or
'current_stream' changes, must change together with 'update_time'*/
  int update_time; /*time of previous update; need to change when 'state' changes, must
change together with 'offset'*/
  int duration;/*duration of the state; need to change when 'state' or 'update_time'
changes */
  struct Client *previous; /*previous client for this movie*/
  struct Client *next; /*next client for this movie*/
}Client;


typedef struct Movie {
  int movieID;
  int duration;
  int multicast_Ct; /*multicast counter*/
  int client_Ct; /*client counter*/
  struct Stream *S_head; /* head of linklist made of multicast streams*/
  struct Stream *S_tail; /* tail of linklist made of multicast streams*/
}Movie;


typedef struct Stream {
  int movieID;
  int launchtime;
  int starttime; /* = launchtime - offset*/
  int offset; /* = play point of the video file when stream launched. 0 <= offset <=
movie duration */
  int clientCt; /*number of clients directly referring to this stream */
  int root; /* whether this stream is root or not */
  int lifetime; /* Suppose the stream plays at regular speed: lifetime = length of the
movie *60 - offset when stream
               has no parent, otherwise lifetime = parent's offset - offset */
  int time_last_kid_merges; /* time when last kid merges in; there is no more pending kid
after that time */
  int ended;
  struct Stream *parent; /* pointer to a Stream; for a multicast stream always points to
previous multicast stream */
  struct Stream *first_kid;
  struct Stream *older_sibling;
  struct Stream *younger_sibling;
  struct Client *C_head; /* head of linklist made of clients*/
  struct Client *C_tail; /* tail of linklist made of clients*/
  int type;
}Stream;

double zipf[NUM_MOVIES];
double theta;           //user: you can change the theta value in function: zipf_set()

int current_sim_start_time;
int no_of_cur_sessions, no_of_cur_channels, no_of_movies, no_of_ended_sessions,
no_of_removed_clients;
int tot_no_of_mem_blks, stop_simu_flag;  //one block = one interval(one second) of movie
int no_of_req_gen, no_of_req_sat, no_of_stream_gen, no_of_ended_streams;
int no_of_cur_multicast_channels, no_of_cur_admission_channels, no_of_cur_i_channels,
no_of_cur_merge_channels;
int no_of_cur_PIC_VCR_sessions;
double abs_time;        //variable to store time
double total_cache_hit, total_session;

TABLE tbl_no_of_cur_sessions, tbl_no_of_cur_channels, tbl_lambda_times_video_duration;
TABLE tbl_session_closeness_to_L_law, tbl_channel_closeness_to_L_law, tbl_chnl_sesn_ratio;
TABLE tbl_no_of_cur_MC_chnls, tbl_no_of_cur_ADM_chnls, tbl_no_of_cur_I_channels,
tbl_no_of_cur_MRGE_channels;
TABLE tbl_no_of_cur_PICVCR_ssns;
```

```
//Movie *m; /* array of movies*/
Movie m[NUM_MOVIES];
void main()
{
      int irun;
      //m = malloc(NUM_MOVIES * (sizeof *m));
      tbl_no_of_cur_sessions = permanent_table("tbl_no_of_cur_sessions"); //initializing
the table
    table_confidence(tbl_no_of_cur_sessions);
    table_run_length(tbl_no_of_cur_sessions, CONF_LOWER, CONF_UPPER, N_RUNS);

      tbl_no_of_cur_channels = permanent_table("tbl_no_of_cur_channels"); //initializing
the table
    table_confidence(tbl_no_of_cur_channels);
    table_run_length(tbl_no_of_cur_channels, CONF_LOWER, CONF_UPPER, N_RUNS);

     tbl_lambda_times_video_duration = permanent_table("tbl_lambda_times_video_duration");
//initializing the table
    table_confidence(tbl_lambda_times_video_duration);
    table_run_length(tbl_lambda_times_video_duration, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_session_closeness_to_L_law =
permanent_table("tbl_session_closeness_to_littles_law"); //initializing the table
    table_confidence(tbl_session_closeness_to_L_law);
    table_run_length(tbl_session_closeness_to_L_law, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_channel_closeness_to_L_law = permanent_table("tbl_channel_closeness_to_L_law");
//initializing the table
    table_confidence(tbl_channel_closeness_to_L_law);
    table_run_length(tbl_channel_closeness_to_L_law, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_chnl_sesn_ratio = permanent_table("tbl_chnl_sesn_ratio"); //initializing the
table
    table_confidence(tbl_chnl_sesn_ratio);
    table_run_length(tbl_chnl_sesn_ratio, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_no_of_cur_MC_chnls = permanent_table("tbl_no_of_cur_MC_chnls"); //initializing
the table
    table_confidence(tbl_no_of_cur_MC_chnls);
    table_run_length(tbl_no_of_cur_MC_chnls, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_no_of_cur_ADM_chnls = permanent_table("tbl_no_of_cur_ADM_chnls"); //initializing
the table
    table_confidence(tbl_no_of_cur_ADM_chnls);
    table_run_length(tbl_no_of_cur_ADM_chnls, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_no_of_cur_I_channels = permanent_table("tbl_no_of_cur_I_channels");
//initializing the table
    table_confidence(tbl_no_of_cur_I_channels);
    table_run_length(tbl_no_of_cur_I_channels, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_no_of_cur_MRGE_channels = permanent_table("tbl_no_of_cur_MRGE_channels");
//initializing the table
    table_confidence(tbl_no_of_cur_MRGE_channels);
    table_run_length(tbl_no_of_cur_MRGE_channels, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_no_of_cur_PICVCR_ssns = permanent_table("tbl_no_of_cur_PICVCR_ssns");
//initializing the table
    table_confidence(tbl_no_of_cur_PICVCR_ssns);
    table_run_length(tbl_no_of_cur_PICVCR_ssns, CONF_LOWER, CONF_UPPER, N_RUNS);

    for (irun = 0; irun <N_RUNS; irun++)  //run simulation
    {
      //sim();
        no_of_cur_sessions = 0;
        no_of_cur_channels = 0;
        no_of_removed_clients = 0;
      //printf(" NO. OF CURRENT CHANNELS  =  %d \n", no_of_cur_channels);
      no_of_req_gen = 0;
      no_of_req_sat = 0;
      no_of_ended_sessions = 0;
      no_of_stream_gen = 0;
      no_of_ended_streams = 0;
      no_of_cur_multicast_channels = 0;
      no_of_cur_admission_channels = 0;
      no_of_cur_i_channels = 0;
      no_of_cur_merge_channels = 0;
      no_of_cur_PIC_VCR_sessions = 0;
        stop_simu_flag = FALSE;
      init_movie(NUM_MOVIES);
      //create("sim");
```

```
      zipf_set();
        req_generator();

        //the results generated
        printf("sim time is %d \n", (int)simtime());
      printf("run No.: %d \n", irun + 1);
      printf("VCR : ");
      if(VERY_INTERACTIVE == 1)
      {
          printf("Very Interactive\n");
      }
      else if(VERY_INTERACTIVE == 0)
      {
          printf("not Very Interactive\n");
      }
      else
      {
          printf("OFF\n");
      }
        if(FIXED_INTERVAL_MULTICAST)
        {
          printf("Fixed Interval Multicasting: ON \n");
         printf(" MULTICAST_INTERVAL = ");
         result_print(MULTICAST_INTERVAL);
        }
      else
      {
          printf("Fixed Interval Multicasting: OFF \n");
      }

      printf(" NO. OF MOVIES  =  ");
        result_print(NUM_MOVIES);
      printf(" INTERARRIVAL TIME  (in seconds) =  ");
      printf(" %f \n",INTERARRIVAL_TIME);
      printf(" DURATION FACTOR f =  ");
      printf(" %f \n",DURATION_FACTOR);
        printf(" CLIENT_BUFFER SIZE  =  ");
        result_print(CLIENT_BUFFER);
      printf(" THETA (skewness: 0 is high, 0.271 is normal, 1 is uniform)  =  ");
        printf(" %f \n",theta);

        printf(" NO. OF REQ GENERATED  =  ");
        result_print(no_of_req_gen);
        printf(" NO. OF REQ SATISFIED  =  ");
        result_print(no_of_req_sat);
        printf(" NO. OF SREAMS USED  =  ");
        result_print(no_of_stream_gen);
        printf(" NO. OF ended_sessions =  ");
        result_print(no_of_ended_sessions);
        printf(" NO. OF ended_streams  =  ");
        result_print(no_of_ended_streams);
        printf(" NO. OF cur_multicast_channels =  ");
        result_print(no_of_cur_multicast_channels);
      printf(" NO. OF cur_admission_channels =  ");
        result_print(no_of_cur_admission_channels);
      printf(" NO. OF cur_i_channels =  ");
        result_print(no_of_cur_i_channels);
      printf(" NO. OF cur_merge_channels =  ");
        result_print(no_of_cur_merge_channels);
      printf(" NO. OF cur_PIC_VCR_sessions = ");
      result_print(no_of_cur_PIC_VCR_sessions);

      printf(" RATIO OF REQ SATISFIED  =  %f %% \n", ((double)(no_of_req_sat *
100)/no_of_req_gen));
        printf(" RATIO OF REQ REJECTED  =  %f %% \n", ((100 - (double)(no_of_req_sat *
100)/no_of_req_gen)));
        printf(" NO. OF CURRENT SESSIONS  =  %d \n", no_of_cur_sessions);
      printf(" NO. OF CURRENT CHANNELS  =  %d \n", no_of_cur_channels);
        printf(" LAMBDA TIMES VIDEO DURATION  =  %f \n",
((double)(IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME)));
        printf(" Closeness Ratio: current session and Little's Law = %f%%  \n",
((double)((no_of_cur_sessions -
IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME)*100/(IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRI
VAL_TIME))));
      printf(" Closeness Ratio: current channels and Little's Law = %f%%  \n",
((double)((no_of_cur_channels -
IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME)*100/(IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRI
VAL_TIME))));
      printf(" Ratio: current channels and current sessions = %f%%  \n\n",
((double)(no_of_cur_channels*100/no_of_cur_sessions)));
```

```
        tabulate(tbl_no_of_cur_sessions, no_of_cur_sessions);
        tabulate(tbl_no_of_cur_channels,no_of_cur_channels);

tabulate(tbl_lambda_times_video_duration,IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME);
      tabulate(tbl_session_closeness_to_L_law,(no_of_cur_sessions -
IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME)*100/(IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRI
VAL_TIME));
      tabulate(tbl_channel_closeness_to_L_law,(no_of_cur_channels -
IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME)*100/(IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRI
VAL_TIME));
        tabulate(tbl_chnl_sesn_ratio,(no_of_cur_channels*100/no_of_cur_sessions));
        tabulate(tbl_no_of_cur_MC_chnls,no_of_cur_multicast_channels);
        tabulate(tbl_no_of_cur_ADM_chnls,no_of_cur_admission_channels);
        tabulate(tbl_no_of_cur_I_channels,no_of_cur_i_channels);
        tabulate(tbl_no_of_cur_MRGE_channels,no_of_cur_merge_channels);
        tabulate(tbl_no_of_cur_PICVCR_ssns,no_of_cur_PIC_VCR_sessions);

      //wait(done);
      //hold(10.0); // time to wait till all the users are done watching their movie
longest movie time
      //        collect_data();

      //        rerun();*/
      //reset();

        //printf(" NO. of streams of movie 1 is %d \n", m[1].multicast_Ct);
        //printf(" NO. of clientes of movie 1 is %d \n", m[1].client_Ct);
    }
    printf(" generating report wait for some time ");
    report();

}

/* function to check whether the disk bandwidth is enough or not*/
int Admission()
{
      return TRUE;
      /*if(!(SESSION_ON_DISK < (MAX_DISK_SESS_POSS - SAFETY)))
       return TRUE;
      else
       return FALSE;*/
}

/* function to keep generating user requests until simulation is over*/
void req_generator()
{
      int mov_id;
      int previous_multicast_start_time = 0;
      current_sim_start_time = (int)simtime();
      while((int)simtime() - current_sim_start_time < S_TIME)
      {
       if(FIXED_INTERVAL_MULTICAST &&
        (previous_multicast_start_time == 0 || (int)simtime() -
previous_multicast_start_time >= MULTICAST_INTERVAL) )
       //launch fixed-interval multicasting stream
       {
        previous_multicast_start_time = (int)simtime();
        for (mov_id = 0; mov_id < NUM_MOVIES; mov_id++)
        {
                Stream *S = New_fixed_interval_multicasting_stream(mov_id);
                stream_Session(S);
        }
       }
       mov_id = choose_movie() - 1;
       if(Admission() == FALSE)
       {
        //printf("request denied due to no channel\n");
        no_of_req_gen++;
       }
       else
       {
        Stream *S = New_stream(mov_id,0); //offset is hardcoded to 0 for the timebeing
        Client *C = New_client(S);

        no_of_req_gen++;
        no_of_req_sat++;

        stream_Session(S);
        user_Session(C);
       }
       //printf(" Time before sleeping in req %f \n", clock);
```

```
       hold(exponential(INTERARRIVAL_TIME)); //expoential distribution for Interarrival
time
       //printf(" Time after waking up in req %f \n", csim_clock);
       if (DEBUG)
       {
        printf("no_of_req_gen is %d \n", no_of_req_gen);
        printf("no_of_cur_sessions is %d \n", no_of_cur_sessions);
        printf("no_of_ended_sessions is %d \n", no_of_ended_sessions);
        printf("current time is %d \n", (int)simtime());
       }
      }
      stop_simu_flag = TRUE;
      hold(60); //for cleanup
}

/* allocate a new stream for a movie and a play point, find a parent if possible*/
struct Stream* New_stream(int movieID, int offset)
{
  Stream* S;
  //channel_capacity--;
  S = malloc(sizeof *S);

  S->movieID = movieID;
  S->offset = offset;
  S->launchtime = (int)simtime();
  S->starttime = S->launchtime - S->offset;
  S->clientCt = 0;
  S->first_kid = NULL;
  S->parent = Parent(S);
  S->older_sibling = NULL;
  S->younger_sibling = NULL;
  S->time_last_kid_merges = 0; //the value 0 is meaningless, just to initialize it here
  S->ended = FALSE;
  S->C_head = NULL;
  S->C_tail = NULL;
  S->type = ADMISSION;

  if (S->parent == NULL)
  {
      printf("no parent found, S offset is %d\n", S->offset);
      S->root = TRUE;
      S->type = MULTICAST;
      S->lifetime = m[S->movieID].duration*60 - S->offset;
      insert_multicast_stream(S);
  }
  else
  {
    if(S->parent->root == TRUE) S->type = MERGE;
    insert_non_multicast_stream(S);
    update_stream(S->parent);
  }

  //stream_Session(S);
  return S;
}

/* allocate a new fixed-interval multicasting stream for a movie and a play point*/
struct Stream*  New_fixed_interval_multicasting_stream(int movieID)
{
  Stream* S;
  //channel_capacity--;
  S = malloc(sizeof *S);

  S->movieID = movieID;
  S->offset = 0;
  S->launchtime = (int)simtime();
  S->starttime = S->launchtime - S->offset;
  S->clientCt = 0;
  S->first_kid = NULL;
  S->parent = NULL;
  S->older_sibling = NULL;
  S->younger_sibling = NULL;
  S->time_last_kid_merges = 0; //the value 0 is meaningless, just to initialize it here
  S->ended = FALSE;
  S->C_head = NULL;
  S->C_tail = NULL;
  S->root = TRUE;
  S->type = FI_MC;
  S->lifetime = m[S->movieID].duration*60 - S->offset;
  insert_multicast_stream(S);
  return S;
```

```
}

/* insert a non multicast stream into its parent stream's kid list*/
void insert_non_multicast_stream(struct Stream* S)
{
      S->root = FALSE;
      S->older_sibling = NULL;
      S->younger_sibling = NULL;
      if(S->first_kid == NULL) //S has no kid stream
      {
       S->lifetime = S->starttime - S->parent->starttime;
      }
      else //S has kid stream(s)
      {
       S->lifetime = S->time_last_kid_merges + (S->starttime - S->parent->starttime) - S-
>launchtime;
      }

      S->older_sibling = Youngest_kid(S->parent);
      if (S->older_sibling != NULL)
      {
       S->older_sibling->younger_sibling = S;
      }
      else
      {
       S->parent->first_kid = S;
        //printf("being the first kid \n");
      }

      if(S->launchtime + S->lifetime > S->parent->time_last_kid_merges)
      {/* update time of the latest mergepoint into S->parent
          if S is its parent's first kid, update will also occur because every new
Stream's initial value
       for time_last_kid_merges was 0 */
        S->parent->time_last_kid_merges = S->launchtime + S->lifetime;
      }
}

/* function to update a stream's new parent after using ERMT */
void update_stream(struct Stream* S)
{
  Stream *new_parent;
  /* the following block should be added if using fixed-periodical multicast method;
     otherwise the multicast stream may merge into another multicast stream

  if (S->root == TRUE) return;
  */

  if(FIXED_INTERVAL_MULTICAST && S->type == FI_MC)
  {
    return;
  }

  /* find a new parent for S using ERMT;
   and this new parent should live longer than time of S' last kid merges into S +
playpiont gap between S and this new parent*/
  new_parent = Parent(S);

  if(new_parent == NULL) //No parent found
  {//extend S' lifetime to the end of the movie, set S as a root;
      S->lifetime = m[S->movieID].duration*60 - S->offset;
      if(S->root != TRUE)
      {//if S was not a root before, insert S into the list of root streams
       if(S->parent != NULL)
       {//if S had a parent, remove the Links in between and update that parent's life if
necessary;
        release_non_multicast_stream(S);
       }
       S->root = TRUE;
         S->type = MULTICAST;
       insert_multicast_stream(S);
      }
      return;
  }
  else //found parent
  {
      if(S->parent != NULL)
    { /*S already has parent, no matter the newly found parent is the same parent we need
to do the following
          because S's life time needs update, the parent's time_last_kid_merges also needs
update*/
```

```
       release_non_multicast_stream(S); //remove the link between S and its parent
      }
      else //S has no parent
      {
       if(S->root == TRUE)
       {
        release_multicast_stream(S);
       }
      }
      S->root = FALSE;                  //actually only needs this if S was root before;
but here instead inside "else" just to be safe
      S->parent = new_parent;           //join new parent
    S->type = MERGE;
      insert_non_multicast_stream(S);  //join new parent's kids list
      update_stream(S->parent);        //update the parent recursively
  }
            if (S->starttime == S->parent->starttime)
            {
                if (S->lifetime == 0 && S->parent->lifetime == 1)
                {
                    printf("bug!!! \n");
                }
                printf("whoho!, there it is. the special case S's starttime == parent's,
S = %X, parent = %X\n", S, S->parent);
            }
}


/* find the youngest kid of a given stream, return NULL if it has no kid */
struct Stream *Youngest_kid(struct Stream *Parent)
{
  Stream *sibling = Parent->first_kid;
  if (sibling == NULL) return NULL;
  else
  {
    while (sibling->younger_sibling != NULL)
    {/*find the youngest kid*/
      sibling = sibling->younger_sibling;
    }
      return sibling;
  }
}

/* find the last client of a given stream, return NULL if it has no client */
struct Client *Last_client(struct Stream *S)
{
  Client *client = S->C_head;
  if (client == NULL) return NULL;
  else
  {
    while (client->next != NULL)
    {/*find the last client*/
      client = client->next;
    }
      return client;
  }
}

/* Insert a multicast stream at the end of link list, which keeps all existing multicast
streams of movie i.
    Will use this function when an unscheduled multicast stream was launched.*/
void add_multicast_stream(struct Stream *S)
{
  if (m[S->movieID].multicast_Ct == 0)
  {
    m[S->movieID].S_head = S;
  }
  else
  {
    m[S->movieID].S_tail->younger_sibling = S;
  }
  S->older_sibling = m[S->movieID].S_tail;
  m[S->movieID].S_tail = S;
  m[S->movieID].multicast_Ct++;
  S->younger_sibling = NULL;
}

/* Insert a multicast stream at the right place of link list, which keeps all existing
multicast streams of movie i
   basing on the new multicast stream's start time */
void insert_multicast_stream(struct Stream *S)
```

```
{
  int loop_counter = 0;
  Stream *closest_root; /* pointer to a Stream */
  S->older_sibling = NULL;
  S->younger_sibling = NULL;
  if (m[S->movieID].S_head == NULL)
  {/* there is not multicast stream in the linklist of the movie yet */
    m[S->movieID].S_head = S;
      m[S->movieID].S_tail = S;
  }
  else
  {
      closest_root = m[S->movieID].S_tail;
    while ((closest_root != NULL) && (S->starttime <= closest_root->starttime) )
      {/* while not found the first reachable multicast stream */
      loop_counter++;
      closest_root = closest_root->older_sibling;
      }


    if (closest_root == NULL) /*not found after going thru the whole list*/
    {
      m[S->movieID].S_head->older_sibling = S;
      S->younger_sibling = m[S->movieID].S_head;
      m[S->movieID].S_head = S;
    }
    else
    {
        if(closest_root->younger_sibling != NULL)
        {
       closest_root->younger_sibling->older_sibling = S;
        }
        else
        {
       m[S->movieID].S_tail = S;
        }
      S->younger_sibling = closest_root->younger_sibling;
      closest_root->younger_sibling = S;
      S->older_sibling = closest_root;
    }
  }
  m[S->movieID].multicast_Ct++;
}


/* void remove_multicast_stream(int movieID) is a function: it removes and release(free)
the
   multicast stream from the link list, which keeps all existing multicast streams of
movie movieID. */
void remove_multicast_stream(struct Stream *S)
{
  if(S->first_kid != NULL)
  {
    Stream *first_kid;
    int time = (int)simtime();
    if(S->time_last_kid_merges == (int)simtime())
    {//S has kid stream(s), and this(these) kid stream(s) merge(s) into S (the kid's life
ends) at the same time when S' life ends
      first_kid = S->first_kid;
      release_kid_streams(S);
      printf("special case happened and handled: kid stream merges into multicast
parent(kid stream life ends) at the moment parent life ends \n");
    }
    else
    {
        printf("There must be error. Function: remove_multicast_stream(struct Stream *S).
But S->first_kid != NULL \n");
        return;
    }
  }
  if (m[S->movieID].multicast_Ct <= 0)
  {/*no multicast of movie movieID*/
      printf("There must be error. Function: remove_multicast_stream(struct Stream *S).
But m[S->movieID].multicast_Ct <= 0 \n");
    return;
  }
  release_multicast_stream(S);
  free(S);
  //channel_capacity++;
}
```

```
/* removes but does not release(free) the multicast stream from the link list, which
keeps all existing multicast
   streams of movie movieID. */
void release_multicast_stream(struct Stream *S)
{
  if (S == m[S->movieID].S_head)
  {/*if it is first client of movie movieID*/
    if (S == m[S->movieID].S_tail) /*or use 'if(S->younger_sibling == NULL)' */
    {/*if it is the only client*/
      m[S->movieID].S_head = NULL;
      m[S->movieID].S_tail = NULL;
    }
    else
    {/*if it is the first but not only client*/
      m[S->movieID].S_head = S->younger_sibling;
      S->younger_sibling->older_sibling = NULL;
    }
  }
  else if (S == m[S->movieID].S_tail) /*or use 'else if(S->younger_sibling == NULL)' */
  { /*if it is last client and not the first(only) client*/
      m[S->movieID].S_tail = S->older_sibling;
      S->older_sibling->younger_sibling = NULL;
  }
  else
  {/*it is not first or last client*/
      S->younger_sibling->older_sibling = S->older_sibling;
      S->older_sibling->younger_sibling = S->younger_sibling;
  }
  S->parent = NULL;
  S->older_sibling = NULL;
  S->younger_sibling = NULL;
  m[S->movieID].multicast_Ct--;
}

/* removes the non multicast stream from the link list, which keeps all existing non
multicast streams of
   the stream's parent stream; release(free) the non multicast stream */
void remove_non_multicast_stream(struct Stream *S)
{
  if (S->root == TRUE)
  {/*never remove a root*/
     printf("There must be error. Function: remove_non_multicast_stream(struct Stream *S).
But S->root == TRUE. \n");
    return;
  }

  if (S->first_kid != NULL)
  {
    Stream *first_kid;
    int time = (int)simtime();
    if(S->time_last_kid_merges == (int)simtime())
    {//S has kid stream(s), and this(these) kid stream(s) merge(s) into S (the kid's life
ends) at the same time when S' life ends
      first_kid = S->first_kid;
      release_kid_streams(S);
      printf("special case happened and handled: kid stream merges into non-multicast
parent(kid stream life ends) at the moment parent life ends(or just slightly earlier)
\n");
    }
    else
    {
        printf("There must be error. Function: remove_non_multicast_stream(struct Stream
*S). But S->first_kid != NULL, = %X \n", S->first_kid);
      return;
    }
  }

  if (S->parent != NULL)
  {//if S has a parent, need to disconnect them
    release_non_multicast_stream(S);
  }
  else//else means S is not a root but has no parent => S is an isolated stream or
Simulation is ended S->parent was set to NULL
  {
      if(S->older_sibling == NULL)
      {
          if(S->younger_sibling != NULL)
          {
              S->younger_sibling->older_sibling = NULL;
          }
      }
```

```
      else //S->older_sibling != NULL
      {
          if(S->younger_sibling == NULL)
          {
              S->older_sibling->younger_sibling = NULL;
          }
          else
          {
              S->older_sibling->younger_sibling = S->younger_sibling;
              S->younger_sibling->older_sibling = S->older_sibling;
          }
      }
      S->older_sibling = NULL;
      S->younger_sibling = NULL;
  }

  free (S);
  //channel_capacity++;
}

/* removes the non multicast stream from the link list, which keeps all existing non
multicast streams of
   the stream's parent stream; does not release(free) the non multicast stream */
void release_non_multicast_stream(struct Stream *S)
{
  if (S->parent->first_kid == S)
  {//if it is first kid
    if (S->younger_sibling == NULL)
    {//if it is the only kid
      S->parent->first_kid = NULL;
    }
    else
    {//if it is the first kid but not the only kid
      S->parent->first_kid = S->younger_sibling;
      S->younger_sibling->older_sibling = NULL;
    }
  }
  else if (S->younger_sibling == NULL)
  {/*if it is last kid and not the first(only) kid*/
    S->older_sibling->younger_sibling = NULL;
  }
  else
  { /*it is not first or last kid*/
      S->younger_sibling->older_sibling = S->older_sibling;
      S->older_sibling->younger_sibling = S->younger_sibling;
  }
  update_time_last_kid_merges(S->parent);
  S->parent = NULL;
  S->older_sibling = NULL;
  S->younger_sibling = NULL;
}

void update_time_last_kid_merges(struct Stream *S)
{
      int time_last_kid_merges = 0;
      struct Stream *temp;

      if(S->first_kid != NULL)
      {
       for(temp = S->first_kid; temp != NULL; temp = temp->younger_sibling)
       {
        if (temp->launchtime + temp->lifetime > time_last_kid_merges)
        {
               time_last_kid_merges = temp->launchtime + temp->lifetime;
        }
       }
      }
      S->time_last_kid_merges = time_last_kid_merges;
}

/* for a given stream, set its kid streams' parent pointer to NULL
   condition is that all these kid streams end at same as given stream ends;
   hence the purpose of this function is to void deadlock problem: parent and kids
happened to end
   at same time, but parent stream was removed first; removing kid streams will cause
complaints.
   When parent and kids streams end at same time, this function will be called when the
parent stream is
   about to be removed, so all the kids' parent point will be pointing to NULL */
void release_kid_streams(struct Stream* S)
{
```

```
  if(S->first_kid != NULL)
  {
    Stream *kid = S->first_kid;
    while (kid != NULL)
    {/*find the last client*/
       if(kid->launchtime + kid->lifetime <= S->launchtime + S->lifetime)
      {
        kid->parent = NULL;
        kid = kid->younger_sibling;
      }
      else
      {
        printf("There must be error. Function: release_kid_streams(struct Stream* S), but
at least one kid doesn't end when S ends. \n");
        return;
      }
    }
    S->first_kid = NULL;
  }
}

/* find a target stream for a given stream */
/*Step 1. Set = {non-I channel x | x's play point > given channel's play point}
If start time < = current time, go to step 2; otherwise go to step 3.
Step 2. Target channel = minimumelement   Set {gap = element's play point - given
channel's play point | (current time + gap) < = element's scheduled ending time} if
target channel is NULL, go to step 4; otherwise do following:
"    gap = target channel's play point - given channel's play point; if gap > minimum
(movie's duration/2, client local buffer size) go to step 4; otherwise: given channel's
ending time = current time + gap; Use Find subroutine (ERMT algorithm) to find a target
channel for target channel, start time = given channel's ending time.
"    return target channel.
Step 3. Target channel = minimumelement   Set {gap = element's play point - given
channel's play point | (start time + gap) < = element's scheduled ending time} if target
channel is NULL, go to step 4; otherwise do following:
"    gap = target channel's play point - given channel's play point; if gap > minimum
(movie's duration/2, client local buffer size) go to step 4; otherwise: given channel's
ending time = start time + gap; Use Find subroutine (ERMT algorithm) to find a target
channel for target channel, start time = given channel's ending time.
"    return target channel.
Step 4. Given channel's ending time = current time + length of the movie - given
channel's play point; return NULL.*/
struct Stream * Parent(struct Stream *S)
{
  /*subject to change*/
  /*Find closest root*/
  Stream *closest_root; /* pointer to a Stream */
  Stream *better_parent; /* pointer to a Stream */
  if (m[S->movieID].S_head == NULL)
  {/* no Stream can satisfy S's request because there is not multicast stream in the
linklist of the movie yet */
    return NULL;
  }
  else if (S->starttime < m[S->movieID].S_head->starttime)
  {/* no Stream can satisfy S's request because S's starttime is earlier than first
multicast stream in the link */
    return NULL;
  }
  closest_root = m[S->movieID].S_tail;
  if (closest_root == NULL)
  {// no Stream can satisfy S's request because there is not multicast stream in the
linklist of the movie yet
    printf("there must be mistake,S_head is not NULL but S_tail is \n");
    //return NULL;
  }
  if(S->first_kid == NULL) //S has no kid stream
  {
    while ( (S->starttime < closest_root->starttime) || S == closest_root
       || (S->starttime - closest_root->starttime > closest_root->launchtime +
closest_root->lifetime - (int)simtime())
      || closest_root->launchtime + closest_root->lifetime < closest_root->starttime +
m[closest_root->movieID].duration*60)
      {// while not found the first reachable multicast stream
       closest_root = closest_root->older_sibling;
       if (closest_root == NULL) return NULL; //not found after going thru the whole list
      }
  }
  else //S has kid stream(s)
  {
     while ( (S->starttime < closest_root->starttime) || S == closest_root
```

```
        || (S->starttime - closest_root->starttime > closest_root->launchtime +
closest_root->lifetime - S->time_last_kid_merges)
      || closest_root->launchtime + closest_root->lifetime < closest_root->starttime +
m[closest_root->movieID].duration*60)
      {// while not found the first reachable multicast stream
      closest_root = closest_root->older_sibling;
        if (closest_root == NULL) return NULL; //not found after going thru the whole list
      }
  }

  /* found a multicast stream, which is reachable */
  if (S->starttime - closest_root->starttime > min(CLIENT_BUFFER, m[S-
>movieID].duration*60/2))
  { /* starttime difference between S' and closest root's is bigger than local buffer can
handle
       Or bigger than half of the movie duration. Then patching is impossible     */
    return NULL;
  }

  better_parent = Better_parent(closest_root, S);
  if (better_parent == NULL)
  {
      //printf ("Found no Better Parent \n");
      //closest_root->clientCt++; /*increment client counter for parent stream, which is
not needed for ERMT*/
    if (closest_root == S)
    {
      printf("closest_root is S itself \n");
      return NULL;
    }
    else
    {
      //if(closest_root > S) printf("closest_root's address, %X, is bigger than S', %X
\n", closest_root, S);
      if (S->starttime == closest_root->starttime)
      {
        printf("whoho!, there it is. the special case S's starttime == closest_root's, S
= %X\n", S);
      }
      return closest_root;
    }
  }
  else
  {
    if (better_parent == S)
    {
      printf("better_parent is S itself \n");
      return NULL;
    }
    else
    {
      //printf ("Found Better Parent \n");
    //better_parent->clientCt++;/*increment client counter for parent stream, which is
not needed for ERMT*/
    //if(better_parent > S) printf("better_parent's address, %X, is bigger than S', %X
\n", better_parent, S);
      return better_parent;
    }
  }
}

/*Try to find better parent. If not found, return closest root as parent
  let Set = set of offsprings of closest_root(passed in as "node");
            where the offsprings include the nodes whose direct parent is the closest
root and
            those of which the closest root is a predecessor
  better_parent = Min of Set {orphan->starttime - element->starttime |
                              ((element->launchtime + element->lifetime - current time) >
(orphan->starttime - element->starttime))
                              && ((orphan->starttime - element->starttime) >= 0) };
    better_parent is address of the element in Set, and the element satisfies as the
closest element to orphan
    subject to: lifetime left of element is longer than what's needed to merge orphan
into element
    make sure orphan's playpoint is behind element's playpoint
*/
struct Stream* Better_parent (struct Stream *node, struct Stream *orphan)
{/* this function is to find a better parent Stream for orphan Stream, and return the
parent's address */
  int gap_a;
  int gap_b;
```

```
  int gap_c;
  int gap_d;

  Stream * parent = NULL;
  Stream * temp = NULL;

  if (node->root == TRUE && node->first_kid == NULL)
  {/*if node is root and has no kid*/
    //printf("node is root and node has no kids \n");
      return NULL;
  }
  if (node->root == TRUE && node->first_kid != NULL)
  {/*if node is root and has kid*/
      //printf("Better_parent recursive call \n");
    return Better_parent(node->first_kid, orphan);
  }


  if(orphan->first_kid == NULL) //if orphan has no kid stream
  {
        gap_a = node->launchtime + node->lifetime - (int)simtime(); //left lifetime of
node
  }
  else //if orphan has kid stream(s)
  {
        gap_a = node->launchtime + node->lifetime - orphan->time_last_kid_merges; //left
lifetime of node
  }
  gap_b = orphan->starttime - node->starttime; //playpoints' different

  //printf(" gap_a is %d,  gap_b is %d \n", gap_a, gap_b);

  if (node->older_sibling != NULL)
  {/*node is not a first kid*/
    if (node->first_kid == NULL)
    {/*node has no kid*/
      if (gap_b > 0 &&  gap_a >= gap_b && node != orphan)
      {/*node itself is qualified as a better parent*/
        return node;
      }
      else
      {
        return NULL;
      }
    }
    else /*node is not a first kid but has kid*/
    {
      parent = Better_parent(node->first_kid, orphan);
      if (parent != NULL) /* found a better parent from current node's offspring.
*/
        {                   /* because parent is a offspring of node, it must be closer to
orphan*/
       return parent;
        }
        else if (gap_b > 0 &&  gap_a >= gap_b && node != orphan)
      {/*node itself is qualified as a better parent*/
       return node;
        }
        else /*node itself is not qualified as a better parent, and     */
        {    /*didn't find a better parent from current node's offspring*/
       return NULL;
        }
    }/*end of else: node is not a first kid, has kid*/
  }
  else /* node is a first kid*/
  { /*4 cases:
        - no kid, no siblings
        - has kid, no siblings
        - no kid, has siblings
        - has kid, has siblings
     */
    if (node->first_kid == NULL)
    { /* node is first kid and has no kid*/
      if (node->younger_sibling == NULL)
      { /* no siblings*/
        if (gap_b > 0 &&  gap_a >= gap_b && node != orphan)
        {/*node itself is qualified as a better parent*/
          return node;
        }
        else
        {
```

```
          return NULL;
        }
      }
      else /*node is first kid, has no kid, has siblings*/
      {
       Stream *sibling;
       if (gap_b > 0 &&  gap_a >= gap_b && node != orphan)
        {/*node itself is qualified as a better parent*/
          parent = node;
          gap_d = gap_b;
        }
        else
        {/*node is not qualified as a better parent*/
          parent = NULL;
        }
        for (sibling = node->younger_sibling; sibling != NULL; sibling = sibling-
>younger_sibling)
        {/*check all siblings of node to find the best parent candidate*/
          temp = Better_parent(sibling, orphan);
          if (temp != NULL)
          {/*found a parent candidate*/
            gap_c =  orphan->starttime - temp->starttime; /* the smaller gap_c, the
closer it's to orphan*/
            if (parent == NULL)
            {//initialize parent variable if it's not initialized by node (node is not a
qualified candidate)
              parent = temp;
              gap_d = gap_c;
            }
            else /* parent != NULL*/
            {//compare with the best candidate found from older_siblings
              if (gap_c < gap_d)
              {
                parent = temp;
                gap_d = gap_c;
              }
            }
          }/*end of if*/
        } /*end of for*/
        return parent; //could be a node or NULL
      }/*end of no kid, has siblings*/
    }/*end of if no kid*/
    else /* node is first kid, has kid*/
    {
      if (node->younger_sibling == NULL)
      { /*node is first kid, has kid, no siblings*/
        parent = Better_parent(node->first_kid, orphan);
        if (parent != NULL) /* found a better parent from current node's offspring.
*/
       {                    /* because parent is a offspring of node, it must be closer to
orphan*/
         return parent;
       }
          else if (gap_b > 0 &&  gap_a >= gap_b && node != orphan)
       {/*node itself is qualified as a better parent*/
         return node;
       }
          else /*node itself is not qualified as a better parent, and     */
       {    /*didn't find a better parent from current node's offspring*/
         return NULL;
       }
      }
      else /*node is first kid, has kid, has siblings*/
      {
       Stream * sibling;
        parent = Better_parent(node->first_kid, orphan);
        if (parent != NULL) /* found a better parent from current node's offspring.
*/
       {                    /* because parent is a offspring of node, it must be closer to
orphan*/
         gap_d = orphan->starttime - parent->starttime;
       }
          else if (gap_b > 0 &&  gap_a >= gap_b && node != orphan)
       {/*node itself is qualified as a better parent*/
         parent = node;
         gap_d = gap_b;
       }
          else /*node itself is not qualified as a better parent, and     */
       {    /*didn't find a better parent from current node's offspring*/
         parent = NULL;
       }
```

```
      for (sibling = node->younger_sibling; sibling != NULL; sibling = sibling-
>younger_sibling)
        {/*check all siblings of node to find the best parent candidate*/
          temp = Better_parent(sibling, orphan);
          if (temp != NULL)
          {/*found a parent candidate*/
            gap_c =  orphan->starttime - temp->starttime;
            if (parent == NULL)
            {/*initialize parent variable if it's not initialized by node
          (Neither node nor any of its offsrpings is qualified candidate)*/
              parent = temp;
              gap_d = gap_c;
            }
            else /* parent != NULL*/
            {//compare with the best candidate found from older_siblings
              if (gap_c < gap_d)
              {
                parent = temp;
                gap_d = gap_c;
              }
            }
          }/*end of if*/
        } /*end of for*/
        return parent;
      } /*end of else: node is first kid, has kid, has siblings'*/
    } /*end of else: node is first kid, has kid*/
  }/*end of else: node is first kid*/
}


/* create a new client*/
struct Client* New_client(struct Stream* S)
{
      Client *C;
      C = malloc(sizeof *C);
      C->movieID = S->movieID;
      C->current_stream = S;
      C->offset = 0;
      C->update_time = (int)simtime();
      C->next = NULL;
      C->previous = NULL;
      C->speed = PLAY_SPEED;
      C->state = PLAY;
      C->duration = (int)exponential(PLAY_Dura);
      add_client(C);
    //user_Session(C);
      return C;
}

/* insert a given client at the end of its current stream's link list, which keeps all
existing clients of the stream.
    Before entering this function, admission must be already granted (which means a new
stream is guaranteed */
void add_client(struct Client *C)
{
  if (C->current_stream->clientCt == 0)
  {
    C->current_stream->C_head = C;
  }
  else
  {
    C->current_stream->C_tail->next = C;
  }
  C->previous = C->current_stream->C_tail;
  C->current_stream->C_tail = C;
  C->current_stream->clientCt++;
  m[C->movieID].client_Ct++;
  C->next = NULL;
}

/* insert an existing client at the end of a given stream's link list, which keeps all
existing clients of the stream.*/
void join_stream(struct Client *C, struct Stream *S)
{
  C->current_stream = S;
  if (S->clientCt == 0)
  {
    S->C_head = C;
  }
  else
```

```
  {
    S->C_tail->next = C;
  }
  C->previous = S->C_tail;
  S->C_tail = C;
  S->clientCt++;
  C->next = NULL;
}
/* remove and release (free) a given client from the client list, which keeps all
existing clients of the watching stream. */
void remove_client(struct Client *C)
{
  if (m[C->movieID].client_Ct <= 0)
  {/*no client of movie movieID*/
      printf("There must be a mistake: client's current movie's client count is less or
equal to 0. \n");
    return;
  }
  release_one_client(C);
  no_of_removed_clients++;
  //printf("a client is removed, no_of_removed_clients is %d \n", no_of_removed_clients);
  m[C->movieID].client_Ct--;
  free(C);
}


void release_one_client(struct Client* C)
{
  if(C->current_stream != NULL)
  {
    if (C->current_stream->clientCt <= 0)
      {
        printf("There must be a mistake: client's current stream's client count is less or
equal to 0. \n");
        return;
      }
    else if (C == C->current_stream->C_head)
      {/*if it is first client of current_stream*/
      if (C == C->current_stream->C_tail) /*or use 'if(C->next == NULL)' */
        {/*if it is the only client*/
        C->current_stream->C_head = NULL;
        C->current_stream->C_tail = NULL;
        }
      else
        {/*if it is the first but not only client*/
        C->current_stream->C_head = C->next;
        C->next->previous = NULL;
        }
      }
    else if (C == C->current_stream->C_tail) /*or use 'else if(C->next == NULL)' */
      { /*if it is last client and not the first(only) client*/
      C->current_stream->C_tail = C->previous;
      C->previous->next = NULL;
      }
    else
      {/*it is not first or last client*/
      C->next->previous = C->previous;
      C->previous->next = C->next;
      }
    C->current_stream->clientCt--;
    //printf("a client is removed, client count is %d \n", C->current_stream->clientCt);

    //change life time of current stream if C was the only client and the current stream
has no kid stream
    if (FIXED_INTERVAL_MULTICAST)//do not change a multicast stream's life time if under
fixed-interval multicasting mode
    {
        if(C->current_stream->clientCt == 0 && C->current_stream->first_kid == NULL && C-
>current_stream->type != FI_MC)
        {
            C->current_stream->lifetime = (int)simtime() - C->current_stream->launchtime;
        }
    }
    else
    {
        if(C->current_stream->clientCt == 0 && C->current_stream->first_kid == NULL)
        {
            C->current_stream->lifetime = (int)simtime() - C->current_stream->launchtime;
        }
    }
  }
```

```
  C->current_stream = NULL;
  C->previous = NULL;
  C->next = NULL;
}

/* function calls zipf_init */
void zipf_set()
{
      //using zipf distribution generate requests
//  int j;
  theta = 0.271; //0 is the highest skewness, 0.271 is normal, 1 is uniform
  zipf_init();
/*  for (j = 1; j <= NUM_MOVIES; j++)
  {
    printf("%2d  %f\t", j, zipf[j]);
    printf("%2d\n", choose_movie());
  } */
}


/* function to generate 4 different lengthes for movies, here each length has 25% chance
   but if a unified movie size, IF_ONLY_ONE_MOVIE_LENGHT, is defined, then only one size
is given to the movies */
void init_movie(int num)
{
      //Using durations of 45, 60, 90 and 120 mins
      int i;
      double prob_val;
      for(i=0;i<num;i++)
      {
       m[i].movieID = i;
       m[i].multicast_Ct = 0;
       m[i].client_Ct = 0;
       m[i].S_head = NULL;
       m[i].S_tail = NULL;

       if (IF_ONLY_ONE_MOVIE_LENGHT) m[i].duration = IF_ONLY_ONE_MOVIE_LENGHT ;
       else
       {
         prob_val = prob();
         if(prob_val < .25)
           m[i].duration = 45;
         else if(prob_val < .50)
           m[i].duration = 60;
         else if(prob_val < .75)
           m[i].duration = 95;
         else
           m[i].duration = 120;
       }
       //movie_last_session_arr[i] = -1;
      }
}

/* this function is to represent a stream session, decide whether the request will be
rejected or not;
   for request accepted, runs simulation and decides when the session should be
terminated */
void stream_Session(struct Stream *S)
{
   int is_new_req, movie_id, killed, request, play_time, debug_counter;
   Stream *first_kid;
   create("new_Stream"); //CHECK its location

   //no_of_cur_sessions++;
   //printf(" NO. OF CURRENT SESSIONS  =  %d \n", curr_no_of_sessions);

   no_of_cur_channels++;
   no_of_stream_gen++;
   movie_id = S->movieID;
   request = TRUE; // if request rejected later due to no memory or disk, change the
value to "FALSE"
   is_new_req = TRUE;
   killed = FALSE;
   //flag to check every second to see whether the current interval-caching is terminated
to release memory or not
   if(Admission() == FALSE)
   {
       request = FALSE;
   }
   play_time = 0;
```

```
   while(((int)simtime() - S->launchtime < S->lifetime) && (request == TRUE) &&
(stop_simu_flag == FALSE)) //sending data
   {
     //printf("I am inside while\n");
     hold(INTERVALDUR);
       play_time++;
     //S->ended++; //debugging
       if (S->root == FALSE && S->parent == NULL && S->clientCt > 1)
       {
        int current_time = (int)simtime();
        printf("%X, clientCT %d, cur_time %d, launch_time %d, finish_time %d \n", S, S-
>clientCt, current_time, S->launchtime, S->launchtime + S->lifetime);
       }
   }
   S->ended = TRUE;

   if (request == TRUE) //only granted request need to do the following. rejected no need
   {
      if(stop_simu_flag == FALSE) // it must be this case: S lifetime ended
      {
       no_of_cur_channels--;
         no_of_ended_streams++;
       if(S->parent != NULL)
       {
        if(S->clientCt > 0)
        {//check whether needs to merge S' clients into parent stream
               if(S->C_head == NULL)
                       printf("Function stream_Session(struct Stream *S) has conflicts, S-
>clientCt > 0, but S->C_head == NULL \n");
               merge_stream(S);
        }
        else
        {// S->clientCt <= 0
                if(FIXED_INTERVAL_MULTICAST)
                {
                    if((S->parent->clientCt == 0) && (S->parent->first_kid == S) && (S-
>younger_sibling == NULL) &&
                        (S->parent->type != FI_MC))
                   {//S->parent stream has no clients and S is S->parent's only kid
                   //need to add one more condition for multicast stream otherwise root
stream might be terminated as well
                           S->parent->lifetime = (int)simtime() - S->parent->launchtime;
                   }
                }
                else
                {
                   if((S->parent->clientCt == 0) && (S->parent->first_kid == S) && (S-
>younger_sibling == NULL))
                   {//S->parent stream has no clients and S is S->parent's only kid
                   //need to add one more condition for multicast stream otherwise root
stream might be terminated as well
                           S->parent->lifetime = (int)simtime() - S->parent->launchtime;
                   }
                }
        }
       }
      }
      else //stop_simu_flag == TRUE
      {
        if(S->root == TRUE)
        {
            no_of_cur_multicast_channels++;
        }
        else
        {
           if (S->type == INTERACTIVE) no_of_cur_i_channels++;
           if (S->type == ADMISSION) no_of_cur_admission_channels++;
           if (S->type == MERGE) no_of_cur_merge_channels++;
           if (S->type == MULTICAST || S->type == FI_MC) printf("BIG BUG, MULTICAST BUT
NOT ROOOOOT?????? \n");
        }
       //if(S->clientCt > 0)
       if(S->first_kid != NULL)
       {/* want to remove S since simulation is over but S still has kids;
           so need to set all kids' "parent" pointer and S' "first_kid" pointer to NULL */
             debug_counter = 0;
        while (S->first_kid != NULL)
        {
                debug_counter++;
               first_kid = S->first_kid->younger_sibling;
               S->first_kid->parent = NULL;
```

```
              S->first_kid->older_sibling = NULL;
              S->first_kid->younger_sibling = NULL;
              S->first_kid = first_kid;
           }
      }
      }
      release_clients(S);


      /* theoretically the following while loop is not needed. But in simulation,
         some clients may end a second or a few seconds later than the stream itself due
to the sync problem.*/
      play_time = 0;
      while(S->clientCt > 0)
      {
        hold(INTERVALDUR);
        play_time++;
        printf("Stream is holding %d second for clients to finish\n", play_time);
        printf("Stream is holding, number of clients is %d\n\n", S->clientCt);
        //printf("Simulation time is up: Stream is holding %d second for clients to
finish\n", play_time);
        //printf("Stream is holding, number of clients is %d\n\n", S->clientCt);
      }

      if (S->root == TRUE)
      {
      remove_multicast_stream(S);
      }
      else
      {
      remove_non_multicast_stream(S);
      }
   }
}

/* for a given stream, sets all its clients' current_stream pointers to its parent */
void merge_stream(struct Stream* S)
{
    int time;
    Stream *kid;
      Client *client = S->C_head;
      if(S->parent == NULL)
      {
       printf("There must be a mistake: stream needs to merge into its parent but its
parentpointer is NULL. \n");
       return;
      }
     else if (S->C_head == NULL)// || S->clientCt == 0) two conditions should be the same,
but put both just in case
      {
      if (S->clientCt != 0) printf("Function merge_stream(struct Stream* S) has conflicts,
S->C_head == NULL, but S->clientCt != 0\n");
      printf("There must be a mistake: stream's clients need to merge into its parent but
stream has no clients . \n");
        return;
      }
      else if (S->first_kid != NULL)
      {
      time = (int)simtime();
      if (S->time_last_kid_merges == (int)simtime())
      {
        printf("merge_stream(), special case handled - kid(s) end(s) at same time(or
slightly earlier than) parent ends. \n");
        kid = S->first_kid;
        while (kid != NULL)
        {
          if(kid->launchtime + kid->lifetime == S->launchtime + S->lifetime)
          {
            merge_stream(kid);
            kid = kid->younger_sibling;
          }
          else
          {
            printf("There must be error. Function: merge_stream(struct Stream* S), but at
least one kid doesn't end when S ends. \n");
            return;
          }
        }
      }
      else
      {
```

```
        time = (int)simtime();
        printf("There must be a mistake: stream's clients need to merge into its parent
but stream still has kid-streams. \n");
        return;
      }
      }

    while (client != NULL)
    {
       client->current_stream = S->parent;
      client = client->next;
    }

    if(S->parent->C_tail != NULL)
      {
        S->C_head->previous = S->parent->C_tail;
        S->parent->C_tail->next = S->C_head;
      }
      else
      {
        S->parent->C_head = S->C_head;
      }
      S->parent->C_tail = S->C_tail;
      S->C_head = NULL;
      S->C_tail = NULL;
      S->first_kid = NULL;
      S->parent->clientCt = S->parent->clientCt + S->clientCt;
      S->clientCt = 0;
}

/* for a given stream, set its all clients' current_stream pointer to NULL*/
void release_clients(struct Stream* S)
{
    if(S->clientCt > 0)
    {
       Client *client = S->C_head;
      while (client != NULL)
      {/*find the last client*/
         client->current_stream = NULL;
        client->previous = NULL;
        client->state = STOP;
        client = client->next;
      }
      S->C_head = NULL;
      S->C_tail = NULL;
      S->clientCt = 0;
    }
}

/* this function is to represent a user session, decide whether the request will be
rejected or not;
   for request accepted, runs simulation and decides when the session should be
terminated */
void user_Session(struct Client *C)
{
   int is_new_req, movie_id, killed, request, play_time, VCR_duration;
   create("user_Session"); //CHECK its location
   no_of_cur_sessions++;
   //printf(" NO. OF CURRENT SESSIONS  =  %d \n", curr_no_of_sessions);
   //no_of_cur_channels++;
   movie_id = C->movieID;
   request = TRUE; // if request rejected later due to no memory or disk, change the
value to "FALSE"
   is_new_req = TRUE;
   killed = FALSE;
   //flag to check every second to see whether the current interval-caching is terminated
to release memory or not
   if(Admission() == FALSE)
   {
        request = FALSE;
   }
   play_time = 0;
   VCR_duration = 0;
   while((C->state != STOP) && (request == TRUE) && stop_simu_flag == FALSE) //sending
data
   {
     //printf("I am inside while\n");
     hold(INTERVALDUR);
     if (VCR_duration == C->duration || C->state == JUMP_F || C->state == JUMP_B)
     {
       check_VCR(C);
```

```
        VCR_duration = 0;
      }
      else
      {
        maintain_VCR(C);
      }
       play_time++;
      VCR_duration++;
    }

    if (request == TRUE) //only granted request need to do the following. rejected no need
    {
       if(stop_simu_flag == FALSE)
       {
       //printf(" NO. OF CURRENT SESSIONS  =  %d \n", curr_no_of_sessions);
       no_of_cur_sessions--;
       //no_of_cur_channels--;
       no_of_ended_sessions++;
        //add_results(no_of_cur_sessions, &total_session); //statistic
       }
      else
      {
          if(C->state == FF || C->state == RW || C->state == SM)
          {
              no_of_cur_PIC_VCR_sessions++;
          }
      }
       remove_client(C);
    }

    //printf("no_of_cur_sessions is %d \n", no_of_cur_sessions);
    //printf("no_of_ended_sessions is %d \n\n", no_of_ended_sessions);
}

/* take a Client pointer, a new state, a new offset and new speed
   update the Client's info*/
void update_client_state(struct Client *C, int new_state, int new_offset, double
new_speed, int new_duration)
{
    switch (C->state) //C->state is current/previous state, while new_state is new/future
state
    {
    case PLAY: /*from PLAY state to another state*/
        {
            if(new_state == JUMP_B || new_state == JUMP_F)
            {/*new_offset is meaningful only if new_state is jump*/
                C->offset = new_offset;
            }
            else
            {/* C->offset needs to be calculated*/
                C->offset = C->offset + (int)simtime() - C->update_time;
            }
            break;
        }
    case JUMP_F:
        {/* from jump forward to play, no need to change offset*/
            if (new_state != PLAY)
            {
                printf("JUMP should only go to state, PLAY instead of %d \n", new_state);
                new_state = PLAY;
            }
        C->offset = new_offset;
            break;
        }
    case JUMP_B:
        {/* from jump backward to play, no need to change offset*/
            if (new_state != PLAY)
            {
                printf("JUMP should only go to state, PLAY instead of %d \n", new_state);
                new_state = PLAY;
            }
        C->offset = new_offset;
            break;
        }
    case STOP:
        {/* from stop to stop, no need to change offset*/
            if (new_state != STOP)
            {
                printf("STOP should not go to any other state such as %d \n", new_state);
                new_state = STOP;
            }
```

```
            break;
        }
    case PAUSE:
        {/* from pause to play, no need to change offset*/
            if (new_state != PLAY)
            {
                printf("PAUSE should only go to state, PLAY instead of %d \n", new_state);
                new_state = PLAY;
            }
            break;
        }
    case FF:
        {/* from fast forward to play, need to calculate offset*/
            if (new_state != PLAY)
            {
                printf("FF should only go to state, PLAY instead of %d \n", new_state);
            }
            C->offset = (int)(C->offset + ((int)simtime() - C->update_time)*C->speed);
            if (C->offset > m[C->movieID].duration*60) C->offset = m[C-
>movieID].duration*60;
            break;
        }
    case RW:
        {/* from rewind to play, need to calculate offset*/
            C->offset = (int)(C->offset + ((int)simtime() - C->update_time)*C->speed);
            if (new_state != PLAY)
            {
                printf("RW should only go to state, PLAY instead of %d. offset is %d \n",
new_state, C->offset);
                new_state = PLAY;
            }
            if (C->offset < 0) C->offset = 0;
            break;
        }
    case SM:
        {/* from slow motion to play, need to calculate offset*/
            if (new_state != PLAY)
            {
                printf("SM should only go to state, PLAY instead of %d \n", new_state);
                new_state = PLAY;
            }
            C->offset = (int)(C->offset + ((int)simtime() - C->update_time)*C->speed);
            if (C->offset > m[C->movieID].duration*60) C->offset = m[C-
>movieID].duration*60;
        if (C->offset < 0) C->offset = 0;
            break;
        }
    }
    C->update_time = (int)simtime();
    C->state = new_state;
    C->duration = new_duration;
    C->speed = new_speed; /*will only be used if the new state is FF, RW or SM; otherwise
only meaningful*/
}

/* return a VCR speed according to given state
    1. if using fixed FF/RW/SM speed as defined, return pre-defined value
    2. if not using fixed FF/RW/SM speed, instead picking speed a from several options,
then use probablity table*/
double VCR_speed(int state)
{
  double prob = prob();
  if (SLOW_MOTION_SPEED && FF_SPEED && RW_SPEED) /* if these speeds are fixed*/
  {
    if (state == SM) return (double)SLOW_MOTION_SPEED;
    else if (state == FF) return (double)FF_SPEED;
    else if (state == RW) return (double)RW_SPEED;
  }
  else
  {
    if (state == SM)
    {
      if (prob <0.5) return (double)SM_B;
        else return (double)SM_F;
    }
    else if (state == FF)
    {
      if (prob <= 0.333) return (double)FF2;
      else if (prob <= 0.666) return (double)FF4;
      else return (double)FF8;
    }
```

```
    else if (state == RW)
    {
      if (prob <= 0.25) return (double)RW1;
      else if (prob <= 0.50) return (double)RW2;
      else if (prob <= 0.75) return (double)RW4;
      else return (double)RW8;
    }
  }
  /*this function only makes sense if state == FF/RW/SM, list other states here just in
case*/
  if (state == PLAY) return (double)PLAY_SPEED;
  else if (state == PAUSE || state == STOP) return (double)IDLE_SPEED;
  else return (double)PLAY_SPEED;
}

/*randomly pick a VCR operation base on given state, state machine chart and probability
table of the chart.
  Return the new state  */
int VCR_pick(int state)
{
  double prob;
  if (state == PLAY)
  {
    prob = prob();
    //according to probability chart, randomly return PLAY/JUMP/PAUSE/PIC_VCR
    //Maybe can return STOP if allowing user to hit stop key during playback
    if (VERY_INTERACTIVE == TRUE) //for very interactive mode test
    {
     if(prob <= 0.46)
      {//stay in PLAY state
        return PLAY;
      }
       else if(prob <= 0.56)
      {//stop
        return STOP;
      }
      else if(prob <= 0.64)
      {//PAUSE
        return PAUSE;
      }
      else if(prob <= 0.72)
      {//Fast Forward w/ picture
        return FF;
      }
      else if(prob <= 0.80)
      {//Rewind w/ picture
        return RW;
      }
      else if(prob <= 0.84)
      {//Slow Motion
        return SM;
      }
      else if(prob <= 0.92)
      {//Jump Forward
        return JUMP_F;
      }
      else
      {//Jump Backward
        return JUMP_B;
      }
    }
    else if(VERY_INTERACTIVE == FALSE)  //for NOT very interactive mode test
    {
      if(prob <= 0.73)
      {//stay in PLAY state
        return PLAY;
      }
       else if(prob <= 0.78)
      {//stop
        return STOP;
      }
      else if(prob <= 0.82)
      {//PAUSE
        return PAUSE;
      }
      else if(prob <= 0.86)
      {//Fast Forward w/ picture
        return FF;
      }
      else if(prob <= 0.90)
      {//Rewind w/ picture
```

```
        return RW;
      }
      else if(prob <= 0.92)
      {//Slow Motion
        return SM;
      }
      else if(prob <= 0.96)
      {//Jump Forward
        return JUMP_F;
      }
      else
      {//Jump Backward
        return JUMP_B;
      }
    }
    else
    {
      return PLAY;
    }
  }
  else if (state == STOP)
  {/*theoretically this case should never be encountered. Put here just in case*/
    return STOP;
  }
  else /* c->state is JUMP, PAUSE or PIC_VCR*/
  {/*according to probability chart, return PLAY*/
    return PLAY;
  }
}

/*base on given client playing state, return a duration*/
int state_duration(int state)
{
  int Duration = 0;
  switch (state)
  {
    case PLAY:
      Duration = (int)exponential(PLAY_Dura);
    case STOP:
      Duration = (int)exponential(STOP_Dura);
    case PAUSE:
      Duration = (int)exponential(PAUSE_Dura);
    case FF:
       Duration = (int)exponential(FFRW_Dura);
    case RW:
       Duration = (int)exponential(FFRW_Dura);
    case SM:
       Duration = (int)exponential(SM_Dura);
    case JUMP_F:
       Duration = (int)exponential(JUMP_Dura);
    case JUMP_B:
       Duration = (int)exponential(JUMP_Dura);
  }
  Duration = (int)(Duration * DURATION_FACTOR);
  if (Duration <= 0) return 1;
  else return Duration;
}


/* randomly return a jump point base on current playpoint and jump direction.
   According to the state machine, previous state before jump must be PLAY. */
int pick_jump_point(struct Client *C, int jump_direction)
{
  //offset means current offset
  int offset = C->offset + (int)simtime() - C->update_time; //assume has been in play
state since last update_time
  double prob = prob();
  if (jump_direction == JUMP_B)
  {/*return a playpoint before C's current offset*/
    return (int)(offset * prob);
  }
  else if (jump_direction == JUMP_F)
  {/*return a playpoint after C's current offset*/
    return (int)(offset + (m[C->movieID].duration*60 - offset) * prob);
  }
  else
  { /*it should never come into this branch. added it here just in case*/
    return offset;
  }
}
```

```
/*check whether given client is supported by local buffer only. Return TRUE or FALSE
   Given client's state is supposed to be PLAY, PAUSE(? must revisit to confirm) or
PIC_VCR */
int on_local_buffer_only(struct Client *C)
{
  if(C->state == STOP || C->state == JUMP_B || C->state == JUMP_F)
  {
    printf("Function: on_local_buffer_only should not be called for state %d \n", C-
>state);
    return FALSE;
  }
  if(C->current_stream == NULL)
  {
    return TRUE;
  }
  else
  {
    return FALSE;
  }
}

/*check whether local buffer can support client's Pic VCR request. Return TRUE or FALSE
  so far only one call used: void pic_VCR(struct Client *C, int new_state).
  Current implementation is based on estimation only because this simulation does record
  local buffer */
int check_local_buffer(struct Client *C, double VCR_speed)
{
      int playpoint = C->offset + (int)simtime() - C->update_time;
      int stream_playpoint;
    if(VCR_speed > 1) //FF
    {
       if(C->current_stream != NULL)
       {
        stream_playpoint = C->current_stream->offset + (int)simtime() - C->current_stream-
>launchtime;
        if(playpoint < stream_playpoint)
        {
                return TRUE;
        }
        else
        {
                return FALSE;
        }
       }
       else //C->current_stream == NULL
       {
        return FALSE;
       }
    }
    else if(VCR_speed > 0) //slow motion forward or play
    {
        if(C->current_stream != NULL)
       {
        return TRUE;
       }
       else
       {
        return FALSE;
       }
    }
    else //RW or SM backward
    {
        return TRUE;
    }
}

/* play()function calls to check whether a playpoint is in client's buffer and the
new_duration can be
entirely supported on local buffer; return TRUE or FALSE*/
int in_buffer(struct Client *C, int playpoint)
{
  int C_latest_playpoint;
  int C_1st_page_DLed_fr_crrnt_strm_prnt;
  int stream_playpoint;
  int stream_parent_playpoint;
  switch (C->state)  //C->state is the previous state, future state should be PLAY
  {//only need to take care of the cases from which will call this function
    case JUMP_B:
       C_latest_playpoint = C->offset + (int)simtime() - C->update_time;
        if(C->offset <= playpoint)
```

```
      {//C's state before jump must be PLAY, C->offset is still at the point when that
PLAY started
       //hence as long as the backup jump is not earlier than the offset it should be in
buffer unless the
           //buffer size limited it.
       if(C->current_stream != NULL)
       {
               stream_playpoint = C->current_stream->offset + (int)simtime() - C-
>current_stream->launchtime;
               if(C->current_stream->parent == NULL)
               {
                       if((int)simtime() - C->update_time <= CLIENT_BUFFER -
(stream_playpoint - C_latest_playpoint) )
                       {//time C played is less or equal to the buffer that can be used,
which is the total buffer size
                        //minus buffer used for saving incoming frames
                               return TRUE;
                       }
                       else
                       {//C->offset is already overwritten in buffer. Thus the earliest
frame can be found is
                        //entirely up to buffer size used for frames that have been played
                               if( playpoint >= C_latest_playpoint - (CLIENT_BUFFER -
(stream_playpoint - C_latest_playpoint)))
                               {//if the jumppoint is later than the earliest frame saved
                                       return TRUE;
                               }
                               else
                               {
                                       return FALSE;
                               }
                       }
               }
               else //C->current_stream has parent, which means another chunk of buffer
is used for incoming videos
               {    //from C->current_stream->parent
                       stream_parent_playpoint = C->current_stream->parent->offset +
(int)simtime() - C->current_stream->parent->launchtime; //wrong?
                       if((int)simtime() - C->update_time <= CLIENT_BUFFER -
(stream_playpoint - C_latest_playpoint)
                                                                       -
(stream_parent_playpoint - stream_playpoint) )
                       {//time C played is less or equal to the buffer that can be used,
which is the total buffer size
                        //minus buffer used for saving incoming frames
                               return TRUE;
                       }
                       else
                       {//C->offset is already overwritten in buffer. Thus the earliest
frame can be found is
                        //entirely up to buffer size used for frames that have been played
                               if( playpoint >= C_latest_playpoint - (CLIENT_BUFFER -
(stream_playpoint - C_latest_playpoint)
                                       - (stream_parent_playpoint - stream_playpoint)))
                               {//if the jumppoint is later than the earliest frame saved
                                       return TRUE;
                               }
                               else
                               {
                                       return FALSE;
                               }
                       }
               }
       }
       else //C->current_stream == NULL
       {
               if((int)simtime() - C->update_time <= CLIENT_BUFFER)
               {// current time minus C->update_time, which is the beginning of the last
PLAY state, is smaller
                // than the buffer size
                       return TRUE;
               }
               else if ( 0 <= C->offset + C->duration - CLIENT_BUFFER <= playpoint)
               {// the jump point is still later than the earliest frame saved in the
buffer
                       return TRUE;
               }
               else // shouldn't be here: buffer cannot support the play longer than
buffer size.
               {//add here just in case
                       return FALSE;
```

```
                }
          }
        }
        else //C->offset > playpoint
        {
         return FALSE;
        }
    case JUMP_F:
         if(C->current_stream != NULL)
        {
         stream_playpoint = C->current_stream->offset + (int)simtime() - C->current_stream-
>launchtime;
         if(playpoint <= stream_playpoint)
         {//jumppoint is earlier than current_stream's current playpoint. Jumppoint is
later than C's latest playpoint by given
          //because it's JUMP_F. Hence the jumppoint must be in buffer
                return TRUE;
         }
         else //playpoint > stream_playpoint, which means jumppoint is later than
current_stream's current playpoint
         {
                if(C->current_stream->parent != NULL)
                {
                       stream_parent_playpoint = C->current_stream->parent->offset +
(int)simtime() - C->current_stream->parent->launchtime;
                       C_1st_page_DLed_fr_crrnt_strm_prnt = C->current_stream->parent-
>offset +
                                                             (C->current_stream->launchtime
+ C->current_stream->lifetime) - //time current stream merges into its parent

C->current_stream->parent->launchtime - //total value now is the playpoint of current
stream's parent's when current stream merges

(stream_parent_playpoint - stream_playpoint); //minus the gap of current stream and its
parent's playpoints
                       if(C_1st_page_DLed_fr_crrnt_strm_prnt <= playpoint <=
stream_parent_playpoint)
                       //playpoint after jump is earlier than current_stream's parent's
playpoint and later than first frame
                       //client downloaded from current_stream's parent. Hence the
playpoint after jump must be in buffer
                       {
                              return TRUE;
                       }
                       else
                       {
                              return FALSE;
                       }
                }
                else //C->current_stream->parent == NULL
                {
                       return FALSE;
                }
         }
        }
        else //C->current_stream == NULL
        {
         if(playpoint < C->offset + C->duration)
         {//C->offset + C->duration is the latest possible known frame in buffer
                return TRUE;
         }
         else
         {
                return FALSE;
         }
        }
    case PAUSE:
         return TRUE;
    case FF:
         return FALSE;
    case RW:
         return TRUE;
      case SM:
        if(C->current_stream != NULL)
        {
         return TRUE;
        }
        else
        {//C->current_stream == NULL
         if(C->speed > 0)
         {
```

```
                return FALSE;
        }
        else //C->speed must be < 0
        {
                return TRUE;
        }
       }
    case PLAY:
        return TRUE;
    case STOP:
        return FALSE;
      default:
       return FALSE;
  }
}

/*sub-function of void play(struct Client *C)*/
void play_from_jump(struct Client *C, int new_duration)
{
    double new_speed = VCR_speed(PLAY);
    Stream *temp;
    int resume_point = pick_jump_point(C, C->state);
    // randomly find a jump point base on playpoint before jump and probability chart
    if (in_buffer(C, resume_point) == TRUE)
    {//found resume point in local buffer, do nothing but update client info
        if(C->current_stream != NULL && C->current_stream->parent != NULL && resume_point
> C->offset)
        { //client refers to a stream and the stream has parent (means the stream is
still catching up with its parent),
          //then jump forward and found resume point in local buffer means this client
can refer to current stream's
          //parent instead of current stream
            temp = C->current_stream->parent;
            temp->clientCt++; //in case parent's clientCt is 1
            leave_current_stream(C);  // leave current stream
            temp->clientCt--; //to reflect the truth
            join_stream(C, temp);
        }
        update_client_state(C, PLAY, resume_point, new_speed, new_duration);
    }
    else //not found resume point in local buffer
    {
        if (Admission() == TRUE || (C->current_stream->clientCt== 1 && C->current_stream-
>first_kid == NULL
            && C->current_stream->root == FALSE))
        {// passed admission or C is the only client referring to C->current_stream
            leave_current_stream(C); //leave current stream if have one.
            //if Admission() was FALSE, it should return TRUE now
            //since C->current_stream must already be relieved
            temp = New_stream(C->movieID, resume_point);
            //temp->type = INTERACTIVE;
            stream_Session(temp);
            if (temp != NULL)
            {//since admission passed, this case is guaranteed, but keep the check here
just in case
                join_stream(C, temp);
                update_client_state(C, PLAY, resume_point, new_speed, new_duration);
            }
            else
            {
                C->state = PLAY;
            }
        }//end if
        else // cannot pass admission
        {//go back to original play status before jump
            C->state = PLAY; //this is the only necessary change to C since C->state from
this line since the only
          //thing changed when C's state became jump
          // C->offset no change since it's still the old one when state was PLAY
          // C->update_time no change since it's still the old one when state was PLAY
         // rejection_VCR++; //terminate request
        }//end of else
    }//end of else
}
/*VCR operation, sub-function of check_VCR(Client *C) */
void play(struct Client *C)
{
  int resume_point;
  double new_speed = VCR_speed(PLAY);
  int new_duration = state_duration(PLAY);
  Stream *temp;
```

```
  switch (C->state) //previous state
  {
    case JUMP_B:  //resume from jump backward
    {
      play_from_jump(C, new_duration);
      break;
    }

    case JUMP_F:  //resume from jump forward
    {
      play_from_jump(C, new_duration);
      break;
    }

    case PAUSE: //resume from pause
    {
      /* 1. if client is still referring to any stream; can continue refer to it
          2. if client is no longer referring to any stream, local buffer local buffer is
supposed to be full. So it can
          support for CLIENT_BUFFER long playback time on local buffer only.
          In either case above, no need to do anything but update client's information */
      //offset's value doesn't matter since the update function doesn't really need it
      update_client_state(C, PLAY, 0, new_speed, new_duration);
      break;
    }

    case STOP:
    {//this case is impossible since STOP state is not able to go to any other state
      printf("There must be something wrong since STOP state is not able to go to any
other state such as PLAY \n");
      break;
    }

    case PLAY:
    {//this case means no change of VCR state because it's from "PLAY" to "PLAY"
      //offset's value doesn't matter since the update function doesn't really need it
      resume_point = find_playpoint(C);
      if(C->current_stream != NULL)
       {
        update_client_state(C, PLAY, resume_point, new_speed, new_duration);
       }
       else
       {
        if (Admission() == TRUE)
        {//no current stream or current stream is not isolated; but passed admission
         temp = New_stream(C->movieID, resume_point);
              //temp->type = INTERACTIVE;
         stream_Session(temp);
         if (temp != NULL)
         {//since admission passed, this case is guaranteed, but keep this check here
just in case
                 join_stream(C, temp);
                 update_client_state(C, PLAY, resume_point, new_speed, new_duration);
         }
         else //since admission passed, this case is impossible, but keep it here just in
case
         { //reject PLAY request
                 stop(C);
                 //  rejection_VCR++; //terminate request
         }
        }//end of if
        else //cannot pass admission
        {// reject PLAY request
         stop(C);
         //rejection_VCR++; //terminate request
        }//end of else
       }
       break;
    }

    default: //resume from VCR with picture
    {
      resume_point = find_playpoint(C);
      if(resume_point < 0 || resume_point >= m[C->movieID].duration*60)
      {
        stop(C);
        break;
      }
      if (in_buffer(C, resume_point) == TRUE)
      { //found resume point in local buffer
        update_client_state(C, PLAY, resume_point, new_speed, new_duration);
```

```
      }
      else //not found resume point in local buffer
      {
        if (C->current_stream != NULL && C->current_stream->root == FALSE &&
         C->current_stream->parent == NULL && C->current_stream->clientCt == 1 &&
         C->current_stream->first_kid == NULL)
        {//if client has an isolated stream, which was used to support PIC_VCR; now use
it for PLAY patching support
          if(C->current_stream->C_head != C || C->current_stream->C_tail != C)
          {
              int time_now = (int)simtime();
         printf("current time is %d, stream %X \n", time_now, C->current_stream);
         printf("Isolated stream is not really isolated\n");
          }
          C->current_stream->movieID = C->movieID;
          C->current_stream->launchtime = (int)simtime();
          C->current_stream->offset = resume_point;
          C->current_stream->starttime = C->current_stream->launchtime - C-
>current_stream->offset;
          C->current_stream->clientCt = 1;
          C->current_stream->root = FALSE;
          C->current_stream->parent = Parent(C->current_stream);
          if (C->current_stream->parent == NULL)
          { // no parent found for C's current_stream; then C's current_stream becomes
root, and being added in to multicast stream list of movieID
            C->current_stream->root = TRUE;
            C->current_stream->type = MULTICAST;
            C->current_stream->older_sibling = NULL;
            C->current_stream->younger_sibling=NULL;
            C->current_stream->first_kid = NULL;
            C->current_stream->lifetime = m[C->current_stream->movieID].duration*60 -
resume_point;
            insert_multicast_stream(C->current_stream);
          }
          else
          { // found parent for C's new current_stream
            if (C->current_stream->parent->root == TRUE) C->current_stream->type = MERGE;
            else C->current_stream->type = ADMISSION;
            C->current_stream->first_kid = NULL;
            insert_non_multicast_stream(C->current_stream);
            update_stream(C->current_stream->parent);

            /*if (C->current_stream->starttime == C->current_stream->parent->starttime)
            {
                printf("whoho!, there it is. the special case C->current_stream starttime
== C->current_stream->parent's, C->current_stream = %X\n", C->current_stream);
            }*/
          }//end else
          update_client_state(C, PLAY, resume_point, new_speed, new_duration);
        }//end if
        else if (Admission() == TRUE || (C->current_stream != NULL && C->current_stream-
>clientCt== 1
            && C->current_stream->first_kid == NULL && C->current_stream->root == FALSE))
        {//no current stream or current stream is not isolated; but passed admission or C
is the only client referring to C->current_stream
          leave_current_stream(C); //leave current stream if have one.
          //if Admission() was FALSE, it should return TRUE now
          //since C->current_stream must already be relieved
          temp = New_stream(C->movieID, resume_point);
          //temp->type = INTERACTIVE;
          stream_Session(temp);
          if (temp != NULL)
          {//since admission passed, this case is guaranteed, but keep this check here
just in case
            join_stream(C, temp);
            update_client_state(C, PLAY, resume_point, new_speed, new_duration);
          }
          else //since admission passed, this case is impossible, but keep it here just
in case
          { //reject PLAY request
            stop(C);
          //  rejection_VCR++; //terminate request
          }
        }//end of else if
        else //no current stream or current stream is in a tree; and cannot pass
admission
        {// reject PLAY request
          stop(C);
          //rejection_VCR++; //terminate request
        }//end of else
      }//end of else
```

```
      break;
    }//end of PIC_VCR case
  }//end of switch
}


/*VCR operation, sub-function of check_VCR(Client *C)
   C->state before entering this function must be PLAY.
   The client which C points to should be removed after
   this stop  */
void stop(struct Client *C)
{ //leave current stream
  int new_state = STOP;
  double new_speed = VCR_speed(new_state); //dummy speed
  int new_duration = state_duration(new_state);
  int new_offset = 0; //dummy offset
  //only jump really needs a new offset, other state's new_offset is just a dummy
argument because update_client_state won't new it
  leave_current_stream(C);
  update_client_state(C, new_state, new_offset, new_speed, new_duration);
}

/*VCR operation, sub-function of check_VCR(Client *C)
   C->state before entering this function must be PLAY */
void pause(struct Client *C)
{ //do not leave current stream yet until local buffer is full, will check in function
check_VCR */
  int new_state = PAUSE;
  double new_speed = VCR_speed(new_state); //dummy speed
  int new_duration = state_duration(new_state);
  int new_offset = 0; //dummy offset
  //only jump really needs a new offset, other state's new_offset is just a dummy
argument because update_client_state won't new it
  update_client_state(C, new_state, new_offset, new_speed, new_duration);
}

/*VCR operation, sub-function of check_VCR(Client *C)
   C->state before entering this function must be PLAY */
void jump(struct Client *C, int new_state)
{
  /*do not leave current stream yet until the client has to, will check when resume */
  //find new speed, new duration and jump point
  /*double new_speed = VCR_speed(new_state); //dummy speed
  int new_duration = state_duration(new_state);
  int new_offset = 0; //dummy offset
  if(new_state == JUMP_B || new_state == JUMP_F)
  {//only jump really needs a new offset
      new_offset = pick_jump_point(C, new_state);
  }
  //only jump really needs a new offset, other state's new_offset is just a dummy
argument because update_client_state won't new it
  update_client_state(C, new_state, new_offset, new_speed, new_duration);*/
  C->state = new_state;
}

/*VCR operation, sub-function of check_VCR(Client *C)
   C->state before entering this function must be PLAY */
void pic_VCR(struct Client *C, int new_state)
{
//find new speed, new duration and jump point(for jump state only)
  double new_speed = VCR_speed(new_state); //real speed
  int new_duration = state_duration(new_state);
  int new_offset = 0; //dummy offset
  Stream *temp;
  //only jump really needs a new offset, other state's new_offset is just a dummy
argument because update_client_state won't new it

  if(check_local_buffer(C, new_speed) == TRUE)
  {/*local buffer can support; so do not leave current stream yet until the client has to;
      will check in function check_VCR*/
    update_client_state(C, new_state, new_offset, new_speed, new_duration);
  }
  else if (Admission() == TRUE ||
  (C->current_stream != NULL && C->current_stream->clientCt == 1
  && C->current_stream->first_kid == NULL && C->current_stream->root == FALSE) )
  {/* local buffer cannot support; but got admission */
    /* allocate an isolated stream to support PIC_VCR*/
    leave_current_stream(C); /*leave current stream if have one. */
    /*if Admission() was FALSE, it should return TRUE now
       since C->current_stream must already be relieved */
    update_client_state(C, new_state, new_offset, new_speed, new_duration);
```

```
    /*get an isolated stream to support current VCR+Picture operation*/
    temp = new_isolated_stream(C);
    join_stream(C, temp);
  }
  else /* cannot pass admission */
  {
     /*reject VCR request, client keeps on PLAY mode*/
     //should be the same as before, added just in case.
      new_duration = state_duration(PLAY);
      update_client_state(C, PLAY, new_offset, PLAY_SPEED, new_duration);
    //rejection_VCR++;
  }
}

/*check if the end of movie is reached while playing or VCR+PIC; return TRUE or FALSE*/
int check_stop(struct Client *C)
{
  if(C->state == JUMP_B || C->state == JUMP_F || C->state == PAUSE)
  {
    return FALSE;
  }
  else if (C->state == STOP)
  {
    return TRUE;
  }
  else if (C->state == PLAY)
  {
    if (C->offset + (int)simtime() - C->update_time >= m[C->movieID].duration*60)
    {
      return TRUE;
    }
    else
    {
      return FALSE;
    }
  }
  else //Pic_VCR: FF, RW, SM
  {
    if ((int)(C->offset + ((int)simtime() - C->update_time)*C->speed) >= m[C-
>movieID].duration*60
        || (int)(C->offset + ((int)simtime() - C->update_time)*C->speed) <= 0 )
    {
      return TRUE;
    }
    else
    {
      return FALSE;
    }
  }
}

/* Only called by function void maintain_VCR(struct Client *C)
played or VCR+PIC on buffer (if state is PIC_VCR, client might or might not still refer
to a stream;
if state is PLAY and client still refers to a stream, there is no need to call this
function; if PLAY on
buffer only, there is need) . This function checks whether the buffer still is able to
support;
return TRUE or FALSE*/
int end_buffer_support(struct Client *C)
{
      int played_time;
      if (C->speed > 0)
      {
       played_time = (int)((simtime() - C->update_time) * C->speed);
      }
      else
      {
       played_time = (int)((simtime() - C->update_time) * C->speed * (-1));
      }

      if( played_time >= exponential(CLIENT_BUFFER))
      {
       return TRUE;
      }
      else
      {
       return FALSE;
      }
}
```

```
/*only for PlAY and VCR_PIC, return current playpoint base on client's previous playpoint,
current time and VCR speed*/
int find_playpoint(struct Client *C)
{
  int offset;
  if(C->state == JUMP_B || C->state == JUMP_F || C->state == PAUSE)
  {
    return C->offset;
  }
  else if (C->state == STOP)
  {
    return m[C->movieID].duration*60;
  }
  else if (C->state == PLAY)
  {
    offset = C->offset + (int)simtime() - C->update_time;
    return offset;
  }
  else //Pic_VCR: FF, RW, SM
  {
    offset = (int)(C->offset + ((int)simtime() - C->update_time)*C->speed);
    return offset;
  }
}

/*check if given client's local buffer is full during PAUSE or PIC_STATE state and
downloading from referred stream;
   return TRUE or FALSE
  so far only called by void maintain_VCR(struct Client *C) */
int local_buffer_full(struct Client *C)
{
  if (C->state == PAUSE || C->state == FF || C->state == RW || C->state == SM)
  {//because no matter what current state is, the download speed is fixed as play speed,
which is 1
    if ((int)simtime() - C->update_time >= CLIENT_BUFFER )
    {
      return TRUE;
    }
    else
    {
      return FALSE;
    }
  }
  else
  {//not suppose to choose this branch at all
      return FALSE;
  }
}

/* given client leaves the stream that was referred*/
void leave_current_stream(struct Client *C)
{
      if (C->current_stream != NULL) // client C has a stream before VCR operation
      {
       release_one_client(C);
      }
}

/* check whether a given client requests a VCR operation and handle it*/
void check_VCR(struct Client *C)
{
  int new_state = VCR_pick(C->state);  /*randomly pick a VCR operation base on current
state*/
  switch (new_state)
  {
    case PLAY: /*play or resume*/
    {
      play(C);
      break;
    }

    /*stop case here can only be triggered if client wants to stop play before the end
      If its probability is 0 in the chart; VCR_pick() will never pick "STOP", which
      means "STOP" can only be reached naturally when movie ends. Natural stop case
      is checked and handled after "else if"*/
    case STOP:
    {
      stop(C);
      break;
    }
```

```
    /* VCR actions w/o pictures: PAUSE and JUMP */

    /*PAUSE, no need to remove the client, no need to drop the current stream right away.
      Time stay in PAUSE varies*/
    case PAUSE:
    {
      pause(C);
      break;
    }

    /*Difference between JUMP and PAUSE is
      1. Time stay in JUMP is always only 1 time because VCR_pick() will pick PLAY when
         input state is JUMP
      2. Afer PAUSE, resume point is fixed; after JUMP, resume point varies*/
    case JUMP_B:
    {
      jump(C, new_state);
      break;
    }

    case JUMP_F:
    {
      jump(C, new_state);
      break;
    }

    /* VCR actions w/ pictures */
    case FF:
    {
      pic_VCR(C, new_state);
      break;
    }

    case RW:
    {
      pic_VCR(C, new_state);
      break;
    }

    case SM:
    {
      pic_VCR(C, new_state);
      break;
    }
  }//end of switch
}//end of check_VCR

void maintain_VCR(struct Client *C)
{
  int playpoint;
  Stream *temp;
  if(C->state == JUMP_F || C->state == JUMP_B)
  {//not suppose to come into this branch because Jump is a state with duration 1. It
never lasts
   //put it here just in case
    return;
  }
  if (C->state == PAUSE)
  {
    if (on_local_buffer_only(C)== FALSE)
    { /*paused but still using current stream to fill out local buffer*/
      if (local_buffer_full(C) == TRUE)
      { /* stop referring to the stream since local buffer is full*/
        leave_current_stream(C);
      }
    }
  }
  else if (C->state == STOP)
  {/*stop case should not be triggered if client's previous state was STOP.
        put here just in case*/
    stop(C);
  }
  else if (check_stop(C) == TRUE)
  {/*state should be PLAY or PIC_VCR, check if the end of movie is reached naturally*/
    stop(C);
  }
  else if (on_local_buffer_only(C)== TRUE && end_buffer_support(C)== TRUE)
  {/*played on buffer only but buffer no longer supports*/
    if (Admission() == TRUE) /*admission control needed to implement*/
    {
      if (C->state == PLAY)
```

```
      {
        playpoint = find_playpoint(C);
        if(playpoint >= m[C->movieID].duration*60) stop(C);
        else
        {
          temp = New_stream(C->movieID, playpoint);
          stream_Session(temp);
          join_stream(C, temp);
        }
      }
      else  /*C->state == PIC_VCR*/
      {
        playpoint = find_playpoint(C);
        if(playpoint < 0 || playpoint >= m[C->movieID].duration*60) stop(C);
        else
        {
          temp = new_isolated_stream(C);
          join_stream(C, temp);
        }
      }
    }
    else
    { /*cannot pass admission*/
      stop(C);
     // rejection_VCR++; /*terminate request*/
    }//end of else
  }//end of else if
  else if ((C->state == FF || C->state == RW || C->state == SM) &&
on_local_buffer_only(C)== FALSE &&
              (C->current_stream->root == TRUE || C->current_stream->parent != NULL) )
  {//state is VCR with picture mode, and not on local buffer only
   //playing on buffer but still downloading from stream which is not an isolated stream
    if (end_buffer_support(C)== TRUE )
    {//buffer can no longer support current VCR operation
      if (Admission() == TRUE ||
          (C->current_stream != NULL && C->current_stream->clientCt == 1
          && C->current_stream->first_kid == NULL && C->current_stream->root == FALSE))
      { //passed admission or C is the only client referring to C->current_stream
        leave_current_stream(C); /*leave current stream if have one. */
        /*if Admission() was FALSE, it should return TRUE now
           since C->current_stream must already be relieved */
        /*get an isolated stream to support current VCR+Picture operation*/
        playpoint = find_playpoint(C);
        if(playpoint < 0 || playpoint >= m[C->movieID].duration*60) stop(C);
        else
        {
          temp = new_isolated_stream(C);
          join_stream(C, temp);
        }
      }
      else
      { //cannot pass admission
        stop(C);
        //rejection_VCR++; //terminate request
      }
    }
    else if (local_buffer_full(C) == TRUE)
    { /* local buffer can still support current VCR operation;
         But stop referring to the stream since local buffer is full*/
      leave_current_stream(C);
    }
  }//end of else if
}

/* allocate a new stream for a movie and a play point without looking for a parent*/
struct Stream * new_isolated_stream(struct Client *C)
{
  Stream* PS;
  //channel_capacity--;
  PS = malloc(sizeof *PS);
  //channel_capacity--;
  PS->movieID = C->movieID;
  PS->offset = find_playpoint(C);
  PS->launchtime = (int)simtime();
  PS->starttime = PS->launchtime - PS->offset;
  PS->clientCt = 0;
  PS->first_kid = NULL;
  PS->parent = NULL;
  PS->root = FALSE;
  PS->older_sibling= NULL;
  PS->younger_sibling=NULL;
```

```
  PS->time_last_kid_merges = 0; //the value 0 is meaningless, just to initialize it here
  PS->ended = FALSE;
  PS->C_head = NULL;
  PS->C_tail = NULL;
  PS->type = INTERACTIVE;
  if(C->state == FF || C->state == SM)
  {
    PS->lifetime = (int)((m[PS->movieID].duration*60 - PS->offset)/C->speed) + 2;
  }
  else if (C->state == RW)
  {
      PS->lifetime = (C->offset/abs((int)C->speed)) + 2;
  }
  stream_Session(PS);
  return PS;
}

/* zipf function from Dr.Kim */
void zipf_init()
{
  double C;
  int i;
  double k;
  double alpha = theta * -1.0;

  for(i=0;i < NUM_MOVIES;i++)
  {
    zipf[i] = 0.0;
  }

  C = 0.0;
  k = 1.0;
  for(i=1 ; i<=NUM_MOVIES ; i++)
  {
    C += pow(1/k, 1.0 + alpha)  ;
    k++;
  } /* for */

  C = 1.0/C;
  k = 1.0;
  for(i=1 ; i <=NUM_MOVIES ; i++)
  {
    zipf[i] = zipf[i-1] + C/pow(k,1.0 + alpha);
    k++;
  } /* for */
} /* zipf_init */

/* CDF function to choose movies using zipf function */
int choose_movie()
{
  double ran_num;
  int i = 1;
  ran_num = prob();
  while(ran_num > zipf[i]) i++;
  return(i);
} /* choose_movie */


//now some verification functions
void result_print(int val)
{
  printf(" %d\n", val);
}

/*add results after each run*/
void add_results(int val, double *my_var)
{
  *my_var =  *my_var + val;
}



/*******************************************************************************************
***
 BEP.h:
 Authors: Ngyat tsui
 Student ID: 002829568
 Project:  CS298 FALL 2008, SJSU
 Instructor: Dr.Suneuy Kim
```

```
************************************************************************************
**/
/* funtion definitions */
void init_movie(int num); //initializing movie array with durations.
void zipf_init();
int choose_movie();
void zipf_set();
void req_generator();
int Admission();
void result_print(int val);
void user_Session(struct Client * S);
void add_client(struct Client *C);
void remove_client(struct Client *C);
void release_one_client(struct Client* C);
void join_stream(struct Client *C, struct Stream *S);
void add_multicast_stream(struct Stream *S);
void insert_multicast_stream(struct Stream *S);
void remove_multicast_stream(struct Stream *S);
void release_multicast_stream(struct Stream *S);
void insert_non_multicast_stream(struct Stream* S);
void remove_non_multicast_stream(struct Stream *S);
void release_non_multicast_stream(struct Stream *S);
void release_kid_streams(struct Stream* S);
void stream_Session(struct Stream *S);
void add_results(int val, double *my_var);
void update_stream(struct Stream* S);
void merge_stream(struct Stream* S);
void release_clients(struct Stream* S);
void update_client_state(struct Client *C, int new_state, int new_offset, double
new_speed, int new_duration);
int on_local_buffer_only(struct Client *C);
int check_local_buffer(struct Client *C, double VCR_speed);
int in_buffer(struct Client *C, int playpoint);
void play(struct Client *C);
void play_from_jump(struct Client *C, int new_duration);
void stop(struct Client *C);
void pause(struct Client *C);
void jump(struct Client *C, int new_state);
void pic_VCR(struct Client *C, int new_state);
struct Stream * new_isolated_stream(struct Client *C);
int find_playpoint(struct Client *C);
void leave_current_stream(struct Client *C);
int local_buffer_full(struct Client *C);
void check_VCR(struct Client *C);
void maintain_VCR(struct Client *C);
int VCR_pick(int state);
double VCR_speed(int state);
int state_duration(int state);
int pick_jump_point(struct Client *C, int jump_direction);
struct Stream* Parent(struct Stream *S);
struct Stream* New_stream(int movieID, int offset);
struct Stream* Youngest_kid(struct Stream *Parent);
struct Client* New_client(struct Stream* S);
struct Stream* Better_parent (struct Stream *node, struct Stream *orphan);
struct Client* Last_client(struct Stream *S);
struct Stream*  New_fixed_interval_multicasting_stream(int movieID);
void update_time_last_kid_merges(struct Stream *S);



/***********************************************************************************
**
 BEP.c:
 Authors: Ngyat tsui
 Student ID: 002829568
 Project:  CS298 FALL 2008, SJSU
 Instructor: Dr.Suneuy Kim

************************************************************************************
**/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "csim.h"
#include "BEP.h"

/*unit for all time period define below is: second*/
#define S_TIME 36000   //user: how long you want to run this program for one time. 9000
on the left means
```

```
                        //      that the simulation will stop when the playbacks of 9000
requests are over
#define N_RUNS 100        //user: how many runs you want for this program
#define MAX_NUM_MOVIES 1200 //don't change this, this is for programers only
#define NUM_MOVIES 1  //user: how many movies are there on the server; please don't
change it bigger than 1200;
                        //zipf needs this too
#define TRUE 1
#define FALSE 0
#define DEBUG 0 //Turn on Debug, set to 1; Turn off Debug, set to 0
#define FIXED_INTERVAL_MULTICAST 1 //Turn on fixed-interval multicast set to 1; Turn off,
set to 0

#define MAXUSERS 4000   //don't change this is for programmers only
#define DISKBW 160         //MBPS, fixed in this simulation, don't change
#define DISPLAYRATE 192       //KBPS MPEG1, fixed in this simulation, don't change
#define INTERVALDUR 1.0 //don't change this is for programmers only
#define SAFETY 20       //don't change this is for programmers only
#define IF_ONLY_ONE_MOVIE_LENGHT 90 //simplified for test in order to avoid hardcoding,
may not need it later

#define INTERARRIVAL_TIME 12.0 //user: the average interarrival time you want, unit =
seconds
#define IC_TRACKING_SIZE 800 //don't change this is for programmers only
#define CONF_LOWER 0.05 //user: the confidence interval you want
#define CONF_UPPER 0.95 //user: the confidence interval you want
#define VERY_INTERACTIVE 1 //VCR test mode: very interactive, set to 1; not very
interactive, set to 0;
                           //set to any other integer to turn off VCR interaction
#define CHANNELS 1000000           /*total channel capacity*/
#define MOVIES  500                     /*total number of movies*/
#define CLIENT_BUFFER  1200     /*client buffer size in terms of play time*/
#define MULTICAST_INTERVAL 120 /*interval between multicast streams in terms of play
time*/
#define DURATION_FACTOR  1.0   /*duration factor used when return a VCR duration   0 <
DURATION_FACTOR <= 2.0*/
#define MULTICAST_DELAY  100   /*delay to close a finished multicast; so done clients can
terminate before the multicast does */

//stream type
#define MULTICAST 0
#define ADMISSION 1
#define INTERACTIVE 2
#define MERGE 3
#define FI_MC 4

/* define state */
#define PLAY 0   /*Playback/Resume*/
#define PAUSE 1    /*Pause*/
#define JUMP_F 2   /* FF without picture (Jump) */
#define JUMP_B 3   /* RW without picture (Jump) */
#define FF 4 /* FF with picture */
#define RW 5 /* RW with picture */
#define SM 6 /* Slow-Motion with picture */
#define STOP 7 /*Stop = done playing*/

/* define state duration */
#define PLAY_Dura 600  /*Playback/Resume*/
#define PAUSE_Dura 300 /*Pause*/
#define JUMP_Dura 1  /* FF/RW without picture (Jump) */
#define FFRW_Dura 150/* FF/RW with picture */
#define SM_Dura 120  /*Slow-Motion with picture*/
#define STOP_Dura 1  /*Stop = done playing*/

/* define state speed */
#define PLAY_SPEED 1  /*regular play speed*/
#define IDLE_SPEED 0  /*speed at pause or stop*/

#define SLOW_MOTION_SPEED 0.5/* fixed slow motion speed*/
#define FF_SPEED 3 /* fixed FF speed*/
#define RW_SPEED -3 /* fixed RW speed*/

#define SM_F 0.5   /*Slow-Motion forward */
#define SM_B -0.5   /*Slow-Motion backward */
#define FF2 2 /*FF x2*/
#define FF4 4 /*FF x4*/
#define FF8 8 /*FF x8*/
#define RW1 -1 /*RW x1*/
#define RW2 -2 /*RW x2*/
#define RW4 -4 /*RW x4*/
#define RW8 -8 /*RW x8*/
```

```
typedef struct Client {
  int movieID;
  struct Stream *current_stream; /* pointer to a Stream */
  int state; /*current play state of the client */
  double speed; /*only for VCR+picture operations (FF, RW or Slow potion) */
  int offset; /*play point of previous update; need to change when 'state' or
'current_stream' changes, must change together with 'update_time'*/
  int update_time; /*time of previous update; need to change when 'state' changes, must
change together with 'offset'*/
  int duration;/*duration of the state; need to change when 'state' or 'update_time'
changes */
  struct Client *previous; /*previous client for this movie*/
  struct Client *next; /*next client for this movie*/
}Client;


typedef struct Movie {
  int movieID;
  int duration;
  int multicast_Ct; /*multicast counter*/
  int client_Ct; /*client counter*/
  struct Stream *S_head; /* head of linklist made of multicast streams*/
  struct Stream *S_tail; /* tail of linklist made of multicast streams*/
}Movie;


typedef struct Stream {
  int movieID;
  int launchtime;
  int starttime; /* = launchtime - offset*/
  int offset; /* = play point of the video file when stream launched. 0 <= offset <=
movie duration */
  int clientCt; /*number of clients directly referring to this stream */
  int root; /* whether this stream is root or not */
  int lifetime; /* Suppose the stream plays at regular speed: lifetime = length of the
movie *60 - offset when stream
               has no parent, otherwise lifetime = parent's offset - offset */
  int time_last_kid_merges; /* time when last kid merges in; there is no more pending kid
after that time */
  int ended;
  struct Stream *parent; /* pointer to a Stream; for a multicast stream always points to
previous multicast stream */
  struct Stream *first_kid;
  struct Stream *older_sibling;
  struct Stream *younger_sibling;
  struct Client *C_head; /* head of linklist made of clients*/
  struct Client *C_tail; /* tail of linklist made of clients*/
  int type;
}Stream;

double zipf[NUM_MOVIES];
double theta;           //user: you can change the theta value in function: zipf_set()

int current_sim_start_time;
int no_of_cur_sessions, no_of_cur_channels, no_of_movies, no_of_ended_sessions,
no_of_removed_clients;
int tot_no_of_mem_blks, stop_simu_flag;  //one block = one interval(one second) of movie
int no_of_req_gen, no_of_req_sat, no_of_stream_gen, no_of_ended_streams;
int no_of_cur_multicast_channels, no_of_cur_admission_channels, no_of_cur_i_channels,
no_of_cur_merge_channels;
int no_of_cur_PIC_VCR_sessions;
double abs_time;        //variable to store time
double total_cache_hit, total_session;

TABLE tbl_no_of_cur_sessions, tbl_no_of_cur_channels, tbl_lambda_times_video_duration;
TABLE tbl_session_closeness_to_L_law, tbl_channel_closeness_to_L_law, tbl_chnl_sesn_ratio;
TABLE tbl_no_of_cur_MC_chnls, tbl_no_of_cur_ADM_chnls, tbl_no_of_cur_I_channels,
tbl_no_of_cur_MRGE_channels;
TABLE tbl_no_of_cur_PICVCR_ssns;

//Movie *m; /* array of movies*/
Movie m[NUM_MOVIES];
void main()
{
      int irun;
      //m = malloc(NUM_MOVIES * (sizeof *m));
      tbl_no_of_cur_sessions = permanent_table("tbl_no_of_cur_sessions"); //initializing
the table
    table_confidence(tbl_no_of_cur_sessions);
    table_run_length(tbl_no_of_cur_sessions, CONF_LOWER, CONF_UPPER, N_RUNS);
```

```
      tbl_no_of_cur_channels = permanent_table("tbl_no_of_cur_channels"); //initializing
the table
    table_confidence(tbl_no_of_cur_channels);
    table_run_length(tbl_no_of_cur_channels, CONF_LOWER, CONF_UPPER, N_RUNS);

     tbl_lambda_times_video_duration = permanent_table("tbl_lambda_times_video_duration");
//initializing the table
    table_confidence(tbl_lambda_times_video_duration);
    table_run_length(tbl_lambda_times_video_duration, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_session_closeness_to_L_law =
permanent_table("tbl_session_closeness_to_littles_law"); //initializing the table
    table_confidence(tbl_session_closeness_to_L_law);
    table_run_length(tbl_session_closeness_to_L_law, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_channel_closeness_to_L_law = permanent_table("tbl_channel_closeness_to_L_law");
//initializing the table
    table_confidence(tbl_channel_closeness_to_L_law);
    table_run_length(tbl_channel_closeness_to_L_law, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_chnl_sesn_ratio = permanent_table("tbl_chnl_sesn_ratio"); //initializing the
table
    table_confidence(tbl_chnl_sesn_ratio);
    table_run_length(tbl_chnl_sesn_ratio, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_no_of_cur_MC_chnls = permanent_table("tbl_no_of_cur_MC_chnls"); //initializing
the table
    table_confidence(tbl_no_of_cur_MC_chnls);
    table_run_length(tbl_no_of_cur_MC_chnls, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_no_of_cur_ADM_chnls = permanent_table("tbl_no_of_cur_ADM_chnls"); //initializing
the table
    table_confidence(tbl_no_of_cur_ADM_chnls);
    table_run_length(tbl_no_of_cur_ADM_chnls, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_no_of_cur_I_channels = permanent_table("tbl_no_of_cur_I_channels");
//initializing the table
    table_confidence(tbl_no_of_cur_I_channels);
    table_run_length(tbl_no_of_cur_I_channels, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_no_of_cur_MRGE_channels = permanent_table("tbl_no_of_cur_MRGE_channels");
//initializing the table
    table_confidence(tbl_no_of_cur_MRGE_channels);
    table_run_length(tbl_no_of_cur_MRGE_channels, CONF_LOWER, CONF_UPPER, N_RUNS);

    tbl_no_of_cur_PICVCR_ssns = permanent_table("tbl_no_of_cur_PICVCR_ssns");
//initializing the table
    table_confidence(tbl_no_of_cur_PICVCR_ssns);
    table_run_length(tbl_no_of_cur_PICVCR_ssns, CONF_LOWER, CONF_UPPER, N_RUNS);

    for (irun = 0; irun <N_RUNS; irun++)  //run simulation
    {
      //sim();
        no_of_cur_sessions = 0;
        no_of_cur_channels = 0;
        no_of_removed_clients = 0;
      //printf(" NO. OF CURRENT CHANNELS  =  %d \n", no_of_cur_channels);
      no_of_req_gen = 0;
      no_of_req_sat = 0;
      no_of_ended_sessions = 0;
      no_of_stream_gen = 0;
      no_of_ended_streams = 0;
      no_of_cur_multicast_channels = 0;
      no_of_cur_admission_channels = 0;
      no_of_cur_i_channels = 0;
      no_of_cur_merge_channels = 0;
      no_of_cur_PIC_VCR_sessions = 0;
        stop_simu_flag = FALSE;
      init_movie(NUM_MOVIES);
      //create("sim");
      zipf_set();
        req_generator();

        //the results generated
        printf("sim time is %d \n", (int)simtime());
      printf("run No.: %d \n", irun + 1);
      printf("VCR : ");
      if(VERY_INTERACTIVE == 1)
      {
          printf("Very Interactive\n");
```

```
      }
      else if(VERY_INTERACTIVE == 0)
      {
         printf("not Very Interactive\n");
      }
      else
      {
         printf("OFF\n");
      }
        if(FIXED_INTERVAL_MULTICAST)
        {
         printf("Fixed Interval Multicasting: ON \n");
        printf(" MULTICAST_INTERVAL = ");
        result_print(MULTICAST_INTERVAL);
        }
      else
      {
         printf("Fixed Interval Multicasting: OFF \n");
      }

      printf(" NO. OF MOVIES  =  ");
        result_print(NUM_MOVIES);
      printf(" INTERARRIVAL TIME  (in seconds) =  ");
      printf(" %f \n",INTERARRIVAL_TIME);
      printf(" DURATION FACTOR f =  ");
      printf(" %f \n",DURATION_FACTOR);
        printf(" CLIENT_BUFFER SIZE  =  ");
        result_print(CLIENT_BUFFER);
      printf(" THETA (skewness: 0 is high, 0.271 is normal, 1 is uniform)  =  ");
        printf(" %f \n",theta);

        printf(" NO. OF REQ GENERATED  =  ");
        result_print(no_of_req_gen);
        printf(" NO. OF REQ SATISFIED  =  ");
        result_print(no_of_req_sat);
        printf(" NO. OF SREAMS USED  =  ");
        result_print(no_of_stream_gen);
        printf(" NO. OF ended_sessions =  ");
        result_print(no_of_ended_sessions);
        printf(" NO. OF ended_streams  =  ");
        result_print(no_of_ended_streams);
        printf(" NO. OF cur_multicast_channels =  ");
        result_print(no_of_cur_multicast_channels);
      printf(" NO. OF cur_admission_channels =  ");
        result_print(no_of_cur_admission_channels);
      printf(" NO. OF cur_i_channels =  ");
        result_print(no_of_cur_i_channels);
      printf(" NO. OF cur_merge_channels =  ");
        result_print(no_of_cur_merge_channels);
      printf(" NO. OF cur_PIC_VCR_sessions = ");
      result_print(no_of_cur_PIC_VCR_sessions);

      printf(" RATIO OF REQ SATISFIED  =  %f %% \n", ((double)(no_of_req_sat *
100)/no_of_req_gen));
        printf(" RATIO OF REQ REJECTED  =  %f %% \n", ((100 - (double)(no_of_req_sat *
100)/no_of_req_gen)));
        printf(" NO. OF CURRENT SESSIONS  =  %d \n", no_of_cur_sessions);
      printf(" NO. OF CURRENT CHANNELS  =  %d \n", no_of_cur_channels);
        printf(" LAMBDA TIMES VIDEO DURATION  =  %f \n",
((double)(IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME)));
        printf(" Closeness Ratio: current session and Little's Law = %f%%  \n",
((double)((no_of_cur_sessions -
IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME)*100/(IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRI
VAL_TIME))));
      printf(" Closeness Ratio: current channels and Little's Law = %f%%  \n",
((double)((no_of_cur_channels -
IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME)*100/(IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRI
VAL_TIME))));
      printf(" Ratio: current channels and current sessions = %f%%  \n\n",
((double)(no_of_cur_channels*100/no_of_cur_sessions)));

        tabulate(tbl_no_of_cur_sessions, no_of_cur_sessions);
        tabulate(tbl_no_of_cur_channels,no_of_cur_channels);

tabulate(tbl_lambda_times_video_duration,IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME);
      tabulate(tbl_session_closeness_to_L_law,(no_of_cur_sessions -
IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME)*100/(IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRI
VAL_TIME));
      tabulate(tbl_channel_closeness_to_L_law,(no_of_cur_channels -
IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRIVAL_TIME)*100/(IF_ONLY_ONE_MOVIE_LENGHT*60/INTERARRI
VAL_TIME));
```

```
        tabulate(tbl_chnl_sesn_ratio,(no_of_cur_channels*100/no_of_cur_sessions));
        tabulate(tbl_no_of_cur_MC_chnls,no_of_cur_multicast_channels);
        tabulate(tbl_no_of_cur_ADM_chnls,no_of_cur_admission_channels);
        tabulate(tbl_no_of_cur_I_channels,no_of_cur_i_channels);
        tabulate(tbl_no_of_cur_MRGE_channels,no_of_cur_merge_channels);
        tabulate(tbl_no_of_cur_PICVCR_ssns,no_of_cur_PIC_VCR_sessions);

      //wait(done);
      //hold(10.0); // time to wait till all the users are done watching their movie
longest movie time
      //       collect_data();

      //       rerun();*/
      //reset();

        //printf(" NO. of streams of movie 1 is %d \n", m[1].multicast_Ct);
        //printf(" NO. of clientes of movie 1 is %d \n", m[1].client_Ct);
    }
    printf(" generating report wait for some time ");
    report();

}

/* function to check whether the disk bandwidth is enough or not*/
int Admission()
{
      return TRUE;
      /*if(!(SESSION_ON_DISK < (MAX_DISK_SESS_POSS - SAFETY)))
       return TRUE;
      else
       return FALSE;*/
}

/* function to keep generating user requests until simulation is over*/
void req_generator()
{
      int mov_id;
      int previous_multicast_start_time = 0;
      current_sim_start_time = (int)simtime();
      while((int)simtime() - current_sim_start_time < S_TIME)
      {
       if(FIXED_INTERVAL_MULTICAST &&
        (previous_multicast_start_time == 0 || (int)simtime() -
previous_multicast_start_time >= MULTICAST_INTERVAL) )
       //launch fixed-interval multicasting stream
       {
        previous_multicast_start_time = (int)simtime();
        for (mov_id = 0; mov_id < NUM_MOVIES; mov_id++)
        {
                Stream *S = New_fixed_interval_multicasting_stream(mov_id);
                stream_Session(S);
        }
         }
       mov_id = choose_movie() - 1;
       if(Admission() == FALSE)
       {
        //printf("request denied due to no channel\n");
        no_of_req_gen++;
       }
       else
       {
        Stream *S = New_stream(mov_id,0); //offset is hardcoded to 0 for the timebeing
        Client *C = New_client(S);

        no_of_req_gen++;
        no_of_req_sat++;

        stream_Session(S);
        user_Session(C);
       }
       //printf(" Time before sleeping in req %f \n", clock);
       hold(exponential(INTERARRIVAL_TIME)); //expoential distribution for Interarrival
time
       //printf(" Time after waking up in req %f \n", csim_clock);
       if (DEBUG)
       {
        printf("no_of_req_gen is %d \n", no_of_req_gen);
        printf("no_of_cur_sessions is %d \n", no_of_cur_sessions);
        printf("no_of_ended_sessions is %d \n", no_of_ended_sessions);
        printf("current time is %d \n", (int)simtime());
       }
```

```
      }
      stop_simu_flag = TRUE;
      hold(60); //for cleanup
}


/* allocate a new stream for a movie and a play point, find a parent if possible*/
struct Stream* New_stream(int movieID, int offset)
{
  Stream* S;
  //channel_capacity--;
  S = malloc(sizeof *S);

  S->movieID = movieID;
  S->offset = offset;
  S->launchtime = (int)simtime();
  S->starttime = S->launchtime - S->offset;
  S->clientCt = 0;
  S->first_kid = NULL;
  S->parent = Parent(S);
  S->older_sibling = NULL;
  S->younger_sibling = NULL;
  S->time_last_kid_merges = 0; //the value 0 is meaningless, just to initialize it here
  S->ended = FALSE;
  S->C_head = NULL;
  S->C_tail = NULL;
  S->type = ADMISSION;

  if (S->parent == NULL)
  {
      printf("no parent found, S offset is %d\n", S->offset);
      S->root = TRUE;
      S->type = MULTICAST;
      S->lifetime = m[S->movieID].duration*60 - S->offset;
      insert_multicast_stream(S);
  }
  else
  {
    if(S->parent->root == TRUE) S->type = MERGE;
    insert_non_multicast_stream(S);
    //update_stream(S->parent);
  }

  //stream_Session(S);
  return S;
}

/* allocate a new fixed-interval multicasting stream for a movie and a play point*/
struct Stream*  New_fixed_interval_multicasting_stream(int movieID)
{
  Stream* S;
  //channel_capacity--;
  S = malloc(sizeof *S);

  S->movieID = movieID;
  S->offset = 0;
  S->launchtime = (int)simtime();
  S->starttime = S->launchtime - S->offset;
  S->clientCt = 0;
  S->first_kid = NULL;
  S->parent = NULL;
  S->older_sibling = NULL;
  S->younger_sibling = NULL;
  S->time_last_kid_merges = 0; //the value 0 is meaningless, just to initialize it here
  S->ended = FALSE;
  S->C_head = NULL;
  S->C_tail = NULL;
  S->root = TRUE;
  S->type = FI_MC;
  S->lifetime = m[S->movieID].duration*60 - S->offset;
  insert_multicast_stream(S);
  return S;
}

/* insert a non multicast stream into its parent stream's kid list*/
void insert_non_multicast_stream(struct Stream* S)
{
      S->root = FALSE;
      S->older_sibling = NULL;
      S->younger_sibling = NULL;
      if(S->first_kid == NULL) //S has no kid stream
```

```
      {
       S->lifetime = S->starttime - S->parent->starttime;
      }
      else //S has kid stream(s)
      {
       S->lifetime = S->time_last_kid_merges + (S->starttime - S->parent->starttime) - S-
>launchtime;
      }

      S->older_sibling = Youngest_kid(S->parent);
      if (S->older_sibling != NULL)
      {
      S->older_sibling->younger_sibling = S;
      }
    else
      {
      S->parent->first_kid = S;
        //printf("being the first kid \n");
      }

    if(S->parent->root == FALSE) //it means S is joining an existing queue instead of a
multicasting stream
    {
        if (S->starttime - S->parent->parent->starttime > S->parent->lifetime - (S-
>launchtime - S->parent->launchtime))
        {
            if(S->parent->parent->root == FALSE)
            {
                printf("it has to be root, otherwise there must be mistakes\n");
            }
            S->parent->lifetime = S->launchtime + S->starttime - S->parent->parent-
>starttime - S->parent->launchtime;
        }
    }
      if(S->launchtime + S->lifetime > S->parent->time_last_kid_merges)
      {/* update time of the latest mergepoint into S->parent
          if S is its parent's first kid, update will also occur because every new
Stream's initial value
       for time_last_kid_merges was 0 */
        S->parent->time_last_kid_merges = S->launchtime + S->lifetime;
      }
}

/* function to update a stream's new parent after using ERMT */
void update_stream(struct Stream* S)
{
  Stream *new_parent;
  /* the following block should be added if using fixed-periodical multicast method;
     otherwise the multicast stream may merge into another multicast stream

  if (S->root == TRUE) return;
  */
  if(FIXED_INTERVAL_MULTICAST && S->type == FI_MC)
  {
    return;
  }

  /* find a new parent for S using ERMT;
   and this new parent should live longer than time of S' last kid merges into S +
playpiont gap between S and this new parent*/
  new_parent = Parent(S);

  if(new_parent == NULL) //No parent found
  {//extend S' lifetime to the end of the movie, set S as a root;
      S->lifetime = m[S->movieID].duration*60 - S->offset;
      if(S->root != TRUE)
      {//if S was not a root before, insert S into the list of root streams
       if(S->parent != NULL)
       {//if S had a parent, remove the Links in between and update that parent's life if
necessary;
        release_non_multicast_stream(S);
       }
       S->root = TRUE;
       S->type = MULTICAST;
       insert_multicast_stream(S);
      }
      return;
  }
  else //found parent
  {
      if(S->parent != NULL)
```

```
    { /*S already has parent, no matter the newly found parent is the same parent we need
to do the following
          because S's life time needs update, the parent's time_last_kid_merges also needs
update*/
       release_non_multicast_stream(S); //remove the link between S and its parent
      }
      else //S has no parent
      {
       if(S->root == TRUE)
       {
        release_multicast_stream(S);
       }
      }
      S->root = FALSE;                  //actually only needs this if S was root before;
but here instead inside "else" just to be safe
      S->parent = new_parent;           //join new parent
      S->type = MERGE;
      insert_non_multicast_stream(S);  //join new parent's kids list
      update_stream(S->parent);        //update the parent recursively
  }
            if (S->starttime == S->parent->starttime)
            {
                if (S->lifetime == 0 && S->parent->lifetime == 1)
                {
                    printf("bug!!! \n");
                }
                printf("whoho!, there it is. the special case S's starttime == parent's,
S = %X, parent = %X\n", S, S->parent);
            }
}


/* find the youngest kid of a given stream, return NULL if it has no kid */
struct Stream *Youngest_kid(struct Stream *Parent)
{
  Stream *sibling = Parent->first_kid;
  if (sibling == NULL) return NULL;
  else
  {
    while (sibling->younger_sibling != NULL)
    {/*find the youngest kid*/
      sibling = sibling->younger_sibling;
    }
     return sibling;
  }
}

/* find the last client of a given stream, return NULL if it has no client */
struct Client *Last_client(struct Stream *S)
{
  Client *client = S->C_head;
  if (client == NULL) return NULL;
  else
  {
    while (client->next != NULL)
    {/*find the last client*/
      client = client->next;
    }
     return client;
  }
}

/* Insert a multicast stream at the end of link list, which keeps all existing multicast
streams of movie i.
    Will use this function when an unscheduled multicast stream was launched.*/
void add_multicast_stream(struct Stream *S)
{
  if (m[S->movieID].multicast_Ct == 0)
  {
    m[S->movieID].S_head = S;
  }
  else
  {
    m[S->movieID].S_tail->younger_sibling = S;
  }
  S->older_sibling = m[S->movieID].S_tail;
  m[S->movieID].S_tail = S;
  m[S->movieID].multicast_Ct++;
  S->younger_sibling = NULL;
}
```

```
/* Insert a multicast stream at the right place of link list, which keeps all existing
multicast streams of movie i
   basing on the new multicast stream's start time */
void insert_multicast_stream(struct Stream *S)
{
  int loop_counter = 0;
  Stream *closest_root; /* pointer to a Stream */
  S->older_sibling = NULL;
  S->younger_sibling = NULL;
  if (m[S->movieID].S_head == NULL)
  {/* there is not multicast stream in the linklist of the movie yet */
    m[S->movieID].S_head = S;
      m[S->movieID].S_tail = S;
  }
  else
  {
      closest_root = m[S->movieID].S_tail;
    while ((closest_root != NULL) && (S->starttime <= closest_root->starttime) )
      {/* while not found the first reachable multicast stream */
      loop_counter++;
      closest_root = closest_root->older_sibling;
      }


    if (closest_root == NULL) /*not found after going thru the whole list*/
    {
      m[S->movieID].S_head->older_sibling = S;
      S->younger_sibling = m[S->movieID].S_head;
      m[S->movieID].S_head = S;
    }
    else
    {
        if(closest_root->younger_sibling != NULL)
        {
       closest_root->younger_sibling->older_sibling = S;
        }
        else
        {
       m[S->movieID].S_tail = S;
        }
      S->younger_sibling = closest_root->younger_sibling;
      closest_root->younger_sibling = S;
      S->older_sibling = closest_root;
    }
  }
  m[S->movieID].multicast_Ct++;
}


/* void remove_multicast_stream(int movieID) is a function: it removes and release(free)
the
   multicast stream from the link list, which keeps all existing multicast streams of
movie movieID. */
void remove_multicast_stream(struct Stream *S)
{
  if(S->first_kid != NULL)
  {
    Stream *first_kid;
    int time = (int)simtime();
    if(S->time_last_kid_merges == (int)simtime())
    {//S has kid stream(s), and this(these) kid stream(s) merge(s) into S (the kid's life
ends) at the same time when S' life ends
      first_kid = S->first_kid;
      release_kid_streams(S);
      printf("special case happened and handled: kid stream merges into multicast
parent(kid stream life ends) at the moment parent life ends \n");
    }
    else
    {
        printf("There must be error. Function: remove_multicast_stream(struct Stream *S).
But S->first_kid != NULL \n");
        return;
    }
  }
  if (m[S->movieID].multicast_Ct <= 0)
  {/*no multicast of movie movieID*/
      printf("There must be error. Function: remove_multicast_stream(struct Stream *S).
But m[S->movieID].multicast_Ct <= 0 \n");
    return;
  }
  release_multicast_stream(S);
```

```
  free(S);
  //channel_capacity++;
}

/* removes but does not release(free) the multicast stream from the link list, which
keeps all existing multicast
   streams of movie movieID. */
void release_multicast_stream(struct Stream *S)
{
  if (S == m[S->movieID].S_head)
  {/*if it is first client of movie movieID*/
    if (S == m[S->movieID].S_tail) /*or use 'if(S->younger_sibling == NULL)' */
    {/*if it is the only client*/
      m[S->movieID].S_head = NULL;
      m[S->movieID].S_tail = NULL;
    }
    else
    {/*if it is the first but not only client*/
      m[S->movieID].S_head = S->younger_sibling;
      S->younger_sibling->older_sibling = NULL;
    }
  }
  else if (S == m[S->movieID].S_tail) /*or use 'else if(S->younger_sibling == NULL)' */
  { /*if it is last client and not the first(only) client*/
      m[S->movieID].S_tail = S->older_sibling;
      S->older_sibling->younger_sibling = NULL;
  }
  else
  {/*it is not first or last client*/
      S->younger_sibling->older_sibling = S->older_sibling;
      S->older_sibling->younger_sibling = S->younger_sibling;
  }
  S->parent = NULL;
  S->older_sibling = NULL;
  S->younger_sibling = NULL;
  m[S->movieID].multicast_Ct--;
}

/* removes the non multicast stream from the link list, which keeps all existing non
multicast streams of
   the stream's parent stream; release(free) the non multicast stream */
void remove_non_multicast_stream(struct Stream *S)
{
  if (S->root == TRUE)
  {/*never remove a root*/
     printf("There must be error. Function: remove_non_multicast_stream(struct Stream *S).
But S->root == TRUE. \n");
    return;
  }

  if (S->first_kid != NULL)
  {
    Stream *first_kid;
    int time = (int)simtime();
    if(S->time_last_kid_merges == (int)simtime())
    {//S has kid stream(s), and this(these) kid stream(s) merge(s) into S (the kid's life
ends) at the same time when S' life ends
      first_kid = S->first_kid;
      release_kid_streams(S);
      printf("special case happened and handled: kid stream merges into non-multicast
parent(kid stream life ends) at the moment parent life ends(or just slightly earlier)
\n");
    }
    else
    {
        printf("There must be error. Function: remove_non_multicast_stream(struct Stream
*S). But S->first_kid != NULL, = %X \n", S->first_kid);
      return;
    }
  }

  if (S->parent != NULL)
  {//if S has a parent, need to disconnect them
    release_non_multicast_stream(S);
  }
  else//else means S is not a root but has no parent => S is an isolated stream or
Simulation is ended S->parent was set to NULL
  {
      if(S->older_sibling == NULL)
      {
          if(S->younger_sibling != NULL)
```

```
          {
              S->younger_sibling->older_sibling = NULL;
          }
      }
      else //S->older_sibling != NULL
      {
          if(S->younger_sibling == NULL)
          {
              S->older_sibling->younger_sibling = NULL;
          }
          else
          {
              S->older_sibling->younger_sibling = S->younger_sibling;
              S->younger_sibling->older_sibling = S->older_sibling;
          }
      }
      S->older_sibling = NULL;
      S->younger_sibling = NULL;
  }

  free (S);
  //channel_capacity++;
}

/* removes the non multicast stream from the link list, which keeps all existing non
multicast streams of
   the stream's parent stream; does not release(free) the non multicast stream */
void release_non_multicast_stream(struct Stream *S)
{
  if (S->parent->first_kid == S)
  {//if it is first kid
    if (S->younger_sibling == NULL)
    {//if it is the only kid
      S->parent->first_kid = NULL;
    }
    else
    {//if it is the first kid but not the only kid
      S->parent->first_kid = S->younger_sibling;
      S->younger_sibling->older_sibling = NULL;
    }
  }
  else if (S->younger_sibling == NULL)
  {/*if it is last kid and not the first(only) kid*/
    S->older_sibling->younger_sibling = NULL;
  }
  else
  { /*it is not first or last kid*/
      S->younger_sibling->older_sibling = S->older_sibling;
      S->older_sibling->younger_sibling = S->younger_sibling;
  }
  update_time_last_kid_merges(S->parent);
  S->parent = NULL;
  S->older_sibling = NULL;
  S->younger_sibling = NULL;
}

void update_time_last_kid_merges(struct Stream *S)
{
      int time_last_kid_merges = 0;
      struct Stream *temp;

      if(S->first_kid != NULL)
      {
       for(temp = S->first_kid; temp != NULL; temp = temp->younger_sibling)
       {
        if (temp->launchtime + temp->lifetime > time_last_kid_merges)
        {
               time_last_kid_merges = temp->launchtime + temp->lifetime;
        }
       }
      }
      S->time_last_kid_merges = time_last_kid_merges;
}

/* for a given stream, set its kid streams' parent pointer to NULL
   condition is that all these kid streams end at same as given stream ends;
   hence the purpose of this function is to void deadlock problem: parent and kids
happened to end
   at same time, but parent stream was removed first; removing kid streams will cause
complaints.
```

```
   When parent and kids streams end at same time, this function will be called when the
parent stream is
   about to be removed, so all the kids' parent point will be pointing to NULL */
void release_kid_streams(struct Stream* S)
{
  if(S->first_kid != NULL)
  {
    Stream *kid = S->first_kid;
    while (kid != NULL)
    {/*find the last client*/
        if(kid->launchtime + kid->lifetime <= S->launchtime + S->lifetime)
      {
        kid->parent = NULL;
        kid = kid->younger_sibling;
      }
      else
      {
        printf("There must be error. Function: release_kid_streams(struct Stream* S), but
at least one kid doesn't end when S ends. \n");
        return;
      }
    }
    S->first_kid = NULL;
  }
}

/* find a target stream for a given stream */
/*Step 1. Set = {non-I channel x | x's play point > given channel's play point}
If start time < = current time, go to step 2; otherwise go to step 3.
Step 2. Target channel = minimumelement   Set {gap = element's play point - given
channel's play point | (current time + gap) < = element's scheduled ending time} if
target channel is NULL, go to step 4; otherwise do following:
"    gap = target channel's play point - given channel's play point; if gap > minimum
(movie's duration/2, client local buffer size) go to step 4; otherwise: given channel's
ending time = current time + gap; Use Find subroutine (ERMT algorithm) to find a target
channel for target channel, start time = given channel's ending time.
"    return target channel.
Step 3. Target channel = minimumelement   Set {gap = element's play point - given
channel's play point | (start time + gap) < = element's scheduled ending time} if target
channel is NULL, go to step 4; otherwise do following:
"    gap = target channel's play point - given channel's play point; if gap > minimum
(movie's duration/2, client local buffer size) go to step 4; otherwise: given channel's
ending time = start time + gap; Use Find subroutine (ERMT algorithm) to find a target
channel for target channel, start time = given channel's ending time.
"    return target channel.
Step 4. Given channel's ending time = current time + length of the movie - given
channel's play point; return NULL.*/
struct Stream * Parent(struct Stream *S)
{
  /*subject to change*/
  /*Find closest root*/
  Stream *closest_root; /* pointer to a Stream */
  Stream *better_parent; /* pointer to a Stream */
  if (m[S->movieID].S_head == NULL)
  {/* no Stream can satisfy S's request because there is not multicast stream in the
linklist of the movie yet */
    return NULL;
  }
  else if (S->starttime < m[S->movieID].S_head->starttime)
  {/* no Stream can satisfy S's request because S's starttime is earlier than first
multicast stream in the link */
    return NULL;
  }
  closest_root = m[S->movieID].S_tail;
  if (closest_root == NULL)
  {// no Stream can satisfy S's request because there is not multicast stream in the
linklist of the movie yet
    printf("there must be mistake,S_head is not NULL but S_tail is \n");
    //return NULL;
  }
  if(S->first_kid == NULL) //S has no kid stream
  {
    while ( (S->starttime < closest_root->starttime) || S == closest_root
        || (S->starttime - closest_root->starttime > closest_root->launchtime +
closest_root->lifetime - (int)simtime())
      || closest_root->launchtime + closest_root->lifetime < closest_root->starttime +
m[closest_root->movieID].duration*60)
      {// while not found the first reachable multicast stream
       closest_root = closest_root->older_sibling;
       if (closest_root == NULL) return NULL; //not found after going thru the whole list
      }
```

```
  }
  else //S has kid stream(s)
  {
      while ( (S->starttime < closest_root->starttime) || S == closest_root
        || (S->starttime - closest_root->starttime > closest_root->launchtime +
closest_root->lifetime - S->time_last_kid_merges)
      || closest_root->launchtime + closest_root->lifetime < closest_root->starttime +
m[closest_root->movieID].duration*60)
      {// while not found the first reachable multicast stream
      closest_root = closest_root->older_sibling;
        if (closest_root == NULL) return NULL; //not found after going thru the whole list
      }
  }

  /* found a multicast stream, which is reachable */
  if (S->starttime - closest_root->starttime > min(CLIENT_BUFFER, m[S-
>movieID].duration*60/2))
  { /* starttime difference between S' and closest root's is bigger than local buffer can
handle
       Or bigger than half of the movie duration. Then patching is impossible     */
    return NULL;
  }

  better_parent = Better_parent(closest_root, S);
  if (better_parent == NULL)
  {
      //printf ("Found no Better Parent \n");
      //closest_root->clientCt++; /*increment client counter for parent stream, which is
not needed for ERMT*/
    if (closest_root == S)
    {
      printf("closest_root is S itself \n");
      return NULL;
    }
    else
    {
      //if(closest_root > S) printf("closest_root's address, %X, is bigger than S', %X
\n", closest_root, S);
      if (S->starttime == closest_root->starttime)
      {
        printf("whoho!, there it is. the special case S's starttime == closest_root's, S
= %X\n", S);
      }
      return closest_root;
    }
  }
  else
  {
    if (better_parent == S)
    {
      printf("better_parent is S itself \n");
      return NULL;
    }
    else
    {
     //printf ("Found Better Parent \n");
    //better_parent->clientCt++;/*increment client counter for parent stream, which is
not needed for ERMT*/
    //if(better_parent > S) printf("better_parent's address, %X, is bigger than S', %X
\n", better_parent, S);
      return better_parent;
    }
  }
}

/*Try to find better parent. If not found, return closest root as parent
  let Set = set of offsprings of closest_root(passed in as "node");
            where the offsprings include the nodes whose direct parent is the closest
root and
            those of which the closest root is a predecessor
  better_parent = Min of Set {orphan->starttime - element->starttime |
                             ((element->launchtime + element->lifetime - current time) >
(orphan->starttime - element->starttime))
                             && ((orphan->starttime - element->starttime) >= 0) };
    better_parent is address of the element in Set, and the element satisfies as the
closest element to orphan
    subject to: lifetime left of element is longer than what's needed to merge orphan
into element
    make sure orphan's playpoint is behind element's playpoint
*/
struct Stream* Better_parent (struct Stream *node, struct Stream *orphan)
```

```
{/* this function is to find a better parent Stream for orphan Stream, and return the
parent's address */
  int gap_a;
  int gap_b;
  int gap_c;
  int gap_d;
  int gap_temp = 0;
  int current_time = (int)simtime();

  Stream * parent = NULL;
  Stream * temp = NULL;
  Stream * existing_queue = NULL;

  if (node->root == TRUE && node->first_kid == NULL)
  {/*if node is root and has no kid*/
    //printf("node is root and node has no kids \n");
      return NULL;
  }

  if(orphan->first_kid == NULL) //if orphan has no kid stream
  {
        gap_a = node->launchtime + node->lifetime - (int)simtime(); //left lifetime of
node
  }
  else //if orphan has kid stream(s)
  {
        gap_a = node->launchtime + node->lifetime - orphan->time_last_kid_merges; //left
lifetime of node
  }
  gap_c = orphan->starttime - node->starttime; //playpoints' different between orphan and
closest root

  existing_queue = node->first_kid;

  while (existing_queue != NULL)
  {
      gap_b = existing_queue->starttime - node->starttime; //playpoints' different
between existing_queue and its parent, the closest root
      if (gap_b <= gap_c && gap_c < gap_b + existing_queue->lifetime - (current_time -
existing_queue->launchtime))
      {
          gap_d = gap_b + existing_queue->lifetime - gap_c - (current_time -
existing_queue->launchtime);
          if (gap_d > gap_b)
          {
              gap_d = gap_b;
          }
          if (gap_d >= gap_temp)
          {
              gap_temp = gap_d;
              temp = existing_queue;
          }
      }

      existing_queue = existing_queue->younger_sibling;
  }
  return temp;
}


/* create a new client*/
struct Client* New_client(struct Stream* S)
{
      Client *C;
      C = malloc(sizeof *C);
      C->movieID = S->movieID;
      C->current_stream = S;
      C->offset = 0;
      C->update_time = (int)simtime();
      C->next = NULL;
      C->previous = NULL;
      C->speed = PLAY_SPEED;
      C->state = PLAY;
      C->duration = (int)exponential(PLAY_Dura);
      add_client(C);
    //user_Session(C);
      return C;
}

/* insert a given client at the end of its current stream's link list, which keeps all
existing clients of the stream.
```

```
    Before entering this function, admission must be already granted (which means a new
stream is guaranteed */
void add_client(struct Client *C)
{
  if (C->current_stream->clientCt == 0)
  {
    C->current_stream->C_head = C;
  }
  else
  {
    C->current_stream->C_tail->next = C;
  }
  C->previous = C->current_stream->C_tail;
  C->current_stream->C_tail = C;
  C->current_stream->clientCt++;
  m[C->movieID].client_Ct++;
  C->next = NULL;
}

/* insert an existing client at the end of a given stream's link list, which keeps all
existing clients of the stream.*/
void join_stream(struct Client *C, struct Stream *S)
{
  C->current_stream = S;
  if (S->clientCt == 0)
  {
    S->C_head = C;
  }
  else
  {
    S->C_tail->next = C;
  }
  C->previous = S->C_tail;
  S->C_tail = C;
  S->clientCt++;
  C->next = NULL;
}
/* remove and release (free) a given client from the client list, which keeps all
existing clients of the watching stream. */
void remove_client(struct Client *C)
{
  if (m[C->movieID].client_Ct <= 0)
  {/*no client of movie movieID*/
     printf("There must be a mistake: client's current movie's client count is less or
equal to 0. \n");
    return;
  }
  release_one_client(C);
  no_of_removed_clients++;
  //printf("a client is removed, no_of_removed_clients is %d \n", no_of_removed_clients);
  m[C->movieID].client_Ct--;
  free(C);
}


void release_one_client(struct Client* C)
{
  if(C->current_stream != NULL)
  {
    if (C->current_stream->clientCt <= 0)
      {
        printf("There must be a mistake: client's current stream's client count is less or
equal to 0. \n");
        return;
      }
    else if (C == C->current_stream->C_head)
      {/*if it is first client of current_stream*/
      if (C == C->current_stream->C_tail) /*or use 'if(C->next == NULL)' */
        {/*if it is the only client*/
        C->current_stream->C_head = NULL;
        C->current_stream->C_tail = NULL;
        }
      else
        {/*if it is the first but not only client*/
        C->current_stream->C_head = C->next;
        C->next->previous = NULL;
        }
      }
    else if (C == C->current_stream->C_tail) /*or use 'else if(C->next == NULL)' */
      { /*if it is last client and not the first(only) client*/
      C->current_stream->C_tail = C->previous;
```

```
      C->previous->next = NULL;
      }
    else
      {/*it is not first or last client*/
      C->next->previous = C->previous;
      C->previous->next = C->next;
      }
    C->current_stream->clientCt--;
    //printf("a client is removed, client count is %d \n", C->current_stream->clientCt);

    //change life time of current stream if C was the only client and the current stream
has no kid stream
    if (FIXED_INTERVAL_MULTICAST)//do not change a multicast stream's life time if under
fixed-interval multicasting mode
    {
        if(C->current_stream->clientCt == 0 && C->current_stream->first_kid == NULL && C-
>current_stream->type != FI_MC)
        {
            C->current_stream->lifetime = (int)simtime() - C->current_stream->launchtime;
        }
    }
    else
    {
        if(C->current_stream->clientCt == 0 && C->current_stream->first_kid == NULL)
        {
            C->current_stream->lifetime = (int)simtime() - C->current_stream->launchtime;
        }
    }
  }
  C->current_stream = NULL;
  C->previous = NULL;
  C->next = NULL;
}

/* function calls zipf_init */
void zipf_set()
{
     //using zipf distribution generate requests
//  int j;
  theta = 0.271; //0 is the highest skewness, 0.271 is normal, 1 is uniform
  zipf_init();
/*  for (j = 1; j <= NUM_MOVIES; j++)
  {
    printf("%2d  %f\t", j, zipf[j]);
    printf("%2d\n", choose_movie());
  } */
}


/* function to generate 4 different lengthes for movies, here each length has 25% chance
   but if a unified movie size, IF_ONLY_ONE_MOVIE_LENGHT, is defined, then only one size
is given to the movies */
void init_movie(int num)
{
      //Using durations of 45, 60, 90 and 120 mins
      int i;
      double prob_val;
      for(i=0;i<num;i++)
      {
       m[i].movieID = i;
       m[i].multicast_Ct = 0;
       m[i].client_Ct = 0;
       m[i].S_head = NULL;
       m[i].S_tail = NULL;

       if (IF_ONLY_ONE_MOVIE_LENGHT) m[i].duration = IF_ONLY_ONE_MOVIE_LENGHT ;
       else
       {
         prob_val = prob();
         if(prob_val < .25)
           m[i].duration = 45;
         else if(prob_val < .50)
           m[i].duration = 60;
         else if(prob_val < .75)
           m[i].duration = 95;
         else
           m[i].duration = 120;
       }
       //movie_last_session_arr[i] = -1;
      }
}
```

```
/* this function is to represent a stream session, decide whether the request will be
rejected or not;
   for request accepted, runs simulation and decides when the session should be
terminated */
void stream_Session(struct Stream *S)
{
   int is_new_req, movie_id, killed, request, play_time, debug_counter;
   Stream *first_kid;
   create("new_Stream"); //CHECK its location

   //no_of_cur_sessions++;
   //printf(" NO. OF CURRENT SESSIONS  =  %d \n", curr_no_of_sessions);

   no_of_cur_channels++;
   no_of_stream_gen++;
   movie_id = S->movieID;
   request = TRUE; // if request rejected later due to no memory or disk, change the
value to "FALSE"
   is_new_req = TRUE;
   killed = FALSE;
   //flag to check every second to see whether the current interval-caching is terminated
to release memory or not
   if(Admission() == FALSE)
   {
      request = FALSE;
   }
   play_time = 0;

   while(((int)simtime() - S->launchtime < S->lifetime) && (request == TRUE) &&
(stop_simu_flag == FALSE)) //sending data
   {
     //printf("I am inside while\n");
     hold(INTERVALDUR);
      play_time++;
     //S->ended++; //debugging
      if (S->root == FALSE && S->parent == NULL && S->clientCt > 1)
      {
       int current_time = (int)simtime();
       printf("%X, clientCT %d, cur_time %d, launch_time %d, finish_time %d \n", S, S-
>clientCt, current_time, S->launchtime, S->launchtime + S->lifetime);
      }
   }
   S->ended = TRUE;

   if (request == TRUE) //only granted request need to do the following. rejected no need
   {
      if(stop_simu_flag == FALSE) // it must be this case: S lifetime ended
      {
       no_of_cur_channels--;
         no_of_ended_streams++;
       if(S->parent != NULL)
       {
        if(S->clientCt > 0)
        {//check whether needs to merge S' clients into parent stream
                if(S->C_head == NULL)
                        printf("Function stream_Session(struct Stream *S) has conflicts, S-
>clientCt > 0, but S->C_head == NULL \n");
                merge_stream(S);
        }
        else
        {// S->clientCt <= 0
                 if(FIXED_INTERVAL_MULTICAST)
                 {
                     if((S->parent->clientCt == 0) && (S->parent->first_kid == S) && (S-
>younger_sibling == NULL) &&
                          (S->parent->type != FI_MC))
                    {//S->parent stream has no clients and S is S->parent's only kid
                    //need to add one more condition for multicast stream otherwise root
stream might be terminated as well
                             S->parent->lifetime = (int)simtime() - S->parent->launchtime;
                    }
                 }
                 else
                 {
                    if((S->parent->clientCt == 0) && (S->parent->first_kid == S) && (S-
>younger_sibling == NULL))
                    {//S->parent stream has no clients and S is S->parent's only kid
                    //need to add one more condition for multicast stream otherwise root
stream might be terminated as well
                             S->parent->lifetime = (int)simtime() - S->parent->launchtime;
```

```
                    }
                }
        }
       }
      }
      else //stop_simu_flag == TRUE
      {
        if(S->root == TRUE)
        {
            no_of_cur_multicast_channels++;
        }
        else
        {
          if (S->type == INTERACTIVE) no_of_cur_i_channels++;
          if (S->type == ADMISSION) no_of_cur_admission_channels++;
          if (S->type == MERGE) no_of_cur_merge_channels++;
          if (S->type == MULTICAST || S->type == FI_MC) printf("BIG BUG, MULTICAST BUT
NOT ROOOOOT?????? \n");
        }
       //if(S->clientCt > 0)
       if(S->first_kid != NULL)
       {/* want to remove S since simulation is over but S still has kids;
           so need to set all kids' "parent" pointer and S' "first_kid" pointer to NULL */
            debug_counter = 0;
        while (S->first_kid != NULL)
        {
                debug_counter++;
               first_kid = S->first_kid->younger_sibling;
               S->first_kid->parent = NULL;
               S->first_kid->older_sibling = NULL;
               S->first_kid->younger_sibling = NULL;
               S->first_kid = first_kid;
            }
       }
       }
       release_clients(S);


       /* theoretically the following while loop is not needed. But in simulation,
          some clients may end a second or a few seconds later than the stream itself due
to the sync problem.*/
       play_time = 0;
       while(S->clientCt > 0)
       {
         hold(INTERVALDUR);
         play_time++;
         printf("Stream is holding %d second for clients to finish\n", play_time);
         printf("Stream is holding, number of clients is %d\n\n", S->clientCt);
         //printf("Simulation time is up: Stream is holding %d second for clients to
finish\n", play_time);
         //printf("Stream is holding, number of clients is %d\n\n", S->clientCt);
       }

       if (S->root == TRUE)
       {
       remove_multicast_stream(S);
       }
       else
       {
       remove_non_multicast_stream(S);
       }
   }
}

/* for a given stream, sets all its clients' current_stream pointers to its parent */
void merge_stream(struct Stream* S)
{
    int time;
    Stream *kid;
      Client *client = S->C_head;
      if(S->parent == NULL)
      {
       printf("There must be a mistake: stream needs to merge into its parent but its
parentpointer is NULL. \n");
       return;
      }
     else if (S->C_head == NULL)// || S->clientCt == 0) two conditions should be the same,
but put both just in case
      {
      if (S->clientCt != 0) printf("Function merge_stream(struct Stream* S) has conflicts,
S->C_head == NULL, but S->clientCt != 0\n");
```

```
      printf("There must be a mistake: stream's clients need to merge into its parent but
stream has no clients . \n");
        return;
      }
      else if (S->first_kid != NULL)
      {
      time = (int)simtime();
      if (S->time_last_kid_merges == (int)simtime())
      {
        printf("merge_stream(), special case handled - kid(s) end(s) at same time(or
slightly earlier than) parent ends. \n");
        kid = S->first_kid;
        while (kid != NULL)
        {
          if(kid->launchtime + kid->lifetime == S->launchtime + S->lifetime)
          {
            merge_stream(kid);
            kid = kid->younger_sibling;
          }
          else
          {
            printf("There must be error. Function: merge_stream(struct Stream* S), but at
least one kid doesn't end when S ends. \n");
            return;
          }
        }
      }
      else
      {
        time = (int)simtime();
        printf("There must be a mistake: stream's clients need to merge into its parent
but stream still has kid-streams. \n");
        return;
      }
      }

    while (client != NULL)
    {
        client->current_stream = S->parent;
      client = client->next;
    }

    if(S->parent->C_tail != NULL)
      {
        S->C_head->previous = S->parent->C_tail;
        S->parent->C_tail->next = S->C_head;
      }
      else
      {
        S->parent->C_head = S->C_head;
      }
      S->parent->C_tail = S->C_tail;
      S->C_head = NULL;
      S->C_tail = NULL;
      S->first_kid = NULL;
      S->parent->clientCt = S->parent->clientCt + S->clientCt;
      S->clientCt = 0;
}

/* for a given stream, set its all clients' current_stream pointer to NULL*/
void release_clients(struct Stream* S)
{
    if(S->clientCt > 0)
    {
        Client *client = S->C_head;
      while (client != NULL)
      {/*find the last client*/
          client->current_stream = NULL;
        client->previous = NULL;
        client->state = STOP;
        client = client->next;
      }
      S->C_head = NULL;
      S->C_tail = NULL;
      S->clientCt = 0;
    }
}

/* this function is to represent a user session, decide whether the request will be
rejected or not;
```

```
   for request accepted, runs simulation and decides when the session should be
terminated */
void user_Session(struct Client *C)
{
   int is_new_req, movie_id, killed, request, play_time, VCR_duration;
   create("user_Session"); //CHECK its location
   no_of_cur_sessions++;
   //printf(" NO. OF CURRENT SESSIONS  =  %d \n", curr_no_of_sessions);
   //no_of_cur_channels++;
   movie_id = C->movieID;
   request = TRUE; // if request rejected later due to no memory or disk, change the
value to "FALSE"
   is_new_req = TRUE;
   killed = FALSE;
   //flag to check every second to see whether the current interval-caching is terminated
to release memory or not
   if(Admission() == FALSE)
   {
        request = FALSE;
   }
   play_time = 0;
   VCR_duration = 0;
   while((C->state != STOP) && (request == TRUE) && stop_simu_flag == FALSE) //sending
data
   {
     //printf("I am inside while\n");
     hold(INTERVALDUR);
     if (VCR_duration == C->duration || C->state == JUMP_F || C->state == JUMP_B)
     {
       check_VCR(C);
       VCR_duration = 0;
     }
     else
     {
       maintain_VCR(C);
     }
       play_time++;
     VCR_duration++;
   }

   if (request == TRUE) //only granted request need to do the following. rejected no need
   {
       if(stop_simu_flag == FALSE)
       {
       //printf(" NO. OF CURRENT SESSIONS  =  %d \n", curr_no_of_sessions);
       no_of_cur_sessions--;
       //no_of_cur_channels--;
       no_of_ended_sessions++;
        //add_results(no_of_cur_sessions, &total_session); //statistic
       }
     else
     {
         if(C->state == FF || C->state == RW || C->state == SM)
         {
             no_of_cur_PIC_VCR_sessions++;
         }
     }
       remove_client(C);
   }

   //printf("no_of_cur_sessions is %d \n", no_of_cur_sessions);
   //printf("no_of_ended_sessions is %d \n\n", no_of_ended_sessions);
}

/* take a Client pointer, a new state, a new offset and new speed
   update the Client's info*/
void update_client_state(struct Client *C, int new_state, int new_offset, double
new_speed, int new_duration)
{
    switch (C->state) //C->state is current/previous state, while new_state is new/future
state
    {
    case PLAY: /*from PLAY state to another state*/
        {
            if(new_state == JUMP_B || new_state == JUMP_F)
            {/*new_offset is meaningful only if new_state is jump*/
                C->offset = new_offset;
            }
            else
            {/* C->offset needs to be calculated*/
                C->offset = C->offset + (int)simtime() - C->update_time;
```

```
            }
            break;
        }
    case JUMP_F:
        {/* from jump forward to play, no need to change offset*/
            if (new_state != PLAY)
            {
                printf("JUMP should only go to state, PLAY instead of %d \n", new_state);
                new_state = PLAY;
            }
        C->offset = new_offset;
            break;
        }
    case JUMP_B:
        {/* from jump backward to play, no need to change offset*/
            if (new_state != PLAY)
            {
                printf("JUMP should only go to state, PLAY instead of %d \n", new_state);
                new_state = PLAY;
            }
        C->offset = new_offset;
            break;
        }
    case STOP:
        {/* from stop to stop, no need to change offset*/
            if (new_state != STOP)
            {
                printf("STOP should not go to any other state such as %d \n", new_state);
                new_state = STOP;
            }
            break;
        }
    case PAUSE:
        {/* from pause to play, no need to change offset*/
            if (new_state != PLAY)
            {
                printf("PAUSE should only go to state, PLAY instead of %d \n", new_state);
                new_state = PLAY;
            }
            break;
        }
    case FF:
        {/* from fast forward to play, need to calculate offset*/
            if (new_state != PLAY)
            {
                printf("FF should only go to state, PLAY instead of %d \n", new_state);
            }
            C->offset = (int)(C->offset + ((int)simtime() - C->update_time)*C->speed);
            if (C->offset > m[C->movieID].duration*60) C->offset = m[C-
>movieID].duration*60;
            break;
        }
    case RW:
        {/* from rewind to play, need to calculate offset*/
            C->offset = (int)(C->offset + ((int)simtime() - C->update_time)*C->speed);
            if (new_state != PLAY)
            {
                printf("RW should only go to state, PLAY instead of %d. offset is %d \n",
new_state, C->offset);
                new_state = PLAY;
            }
            if (C->offset < 0) C->offset = 0;
            break;
        }
    case SM:
        {/* from slow motion to play, need to calculate offset*/
            if (new_state != PLAY)
            {
                printf("SM should only go to state, PLAY instead of %d \n", new_state);
                new_state = PLAY;
            }
            C->offset = (int)(C->offset + ((int)simtime() - C->update_time)*C->speed);
            if (C->offset > m[C->movieID].duration*60) C->offset = m[C-
>movieID].duration*60;
        if (C->offset < 0) C->offset = 0;
            break;
        }
    }
    C->update_time = (int)simtime();
    C->state = new_state;
    C->duration = new_duration;
```

```
    C->speed = new_speed; /*will only be used if the new state is FF, RW or SM; otherwise
only meaningful*/
}

/* return a VCR speed according to given state
    1. if using fixed FF/RW/SM speed as defined, return pre-defined value
    2. if not using fixed FF/RW/SM speed, instead picking speed a from several options,
then use probablity table*/
double VCR_speed(int state)
{
  double prob = prob();
  if (SLOW_MOTION_SPEED && FF_SPEED && RW_SPEED) /* if these speeds are fixed*/
  {
    if (state == SM) return (double)SLOW_MOTION_SPEED;
    else if (state == FF) return (double)FF_SPEED;
    else if (state == RW) return (double)RW_SPEED;
  }
  else
  {
    if (state == SM)
    {
      if (prob <0.5) return (double)SM_B;
        else return (double)SM_F;
    }
    else if (state == FF)
    {
      if (prob <= 0.333) return (double)FF2;
      else if (prob <= 0.666) return (double)FF4;
      else return (double)FF8;
    }
    else if (state == RW)
    {
      if (prob <= 0.25) return (double)RW1;
      else if (prob <= 0.50) return (double)RW2;
      else if (prob <= 0.75) return (double)RW4;
      else return (double)RW8;
    }
  }
  /*this function only makes sense if state == FF/RW/SM, list other states here just in
case*/
  if (state == PLAY) return (double)PLAY_SPEED;
  else if (state == PAUSE || state == STOP) return (double)IDLE_SPEED;
  else return (double)PLAY_SPEED;
}

/*randomly pick a VCR operation base on given state, state machine chart and probability
table of the chart.
  Return the new state  */
int VCR_pick(int state)
{
  double prob;
  if (state == PLAY)
  {
    prob = prob();
    //according to probability chart, randomly return PLAY/JUMP/PAUSE/PIC_VCR
    //Maybe can return STOP if allowing user to hit stop key during playback
    if (VERY_INTERACTIVE == TRUE) //for very interactive mode test
    {
     if(prob <= 0.46)
      {//stay in PLAY state
        return PLAY;
      }
        else if(prob <= 0.56)
      {//stop
        return STOP;
      }
      else if(prob <= 0.64)
      {//PAUSE
        return PAUSE;
      }
      else if(prob <= 0.72)
      {//Fast Forward w/ picture
        return FF;
      }
      else if(prob <= 0.80)
      {//Rewind w/ picture
        return RW;
      }
      else if(prob <= 0.84)
      {//Slow Motion
        return SM;
```

```
      }
      else if(prob <= 0.92)
      {//Jump Forward
        return JUMP_F;
      }
      else
      {//Jump Backward
        return JUMP_B;
      }
    }
    else if(VERY_INTERACTIVE == FALSE)  //for NOT very interactive mode test
    {
      if(prob <= 0.73)
      {//stay in PLAY state
        return PLAY;
      }
       else if(prob <= 0.78)
      {//stop
        return STOP;
      }
      else if(prob <= 0.82)
      {//PAUSE
        return PAUSE;
      }
      else if(prob <= 0.86)
      {//Fast Forward w/ picture
        return FF;
      }
      else if(prob <= 0.90)
      {//Rewind w/ picture
        return RW;
      }
      else if(prob <= 0.92)
      {//Slow Motion
        return SM;
      }
      else if(prob <= 0.96)
      {//Jump Forward
        return JUMP_F;
      }
      else
      {//Jump Backward
        return JUMP_B;
      }
    }
    else
    {
      return PLAY;
    }
  }
  else if (state == STOP)
  {/*theoretically this case should never be encountered. Put here just in case*/
    return STOP;
  }
  else /* c->state is JUMP, PAUSE or PIC_VCR*/
  {/*according to probability chart, return PLAY*/
    return PLAY;
  }
}

/*base on given client playing state, return a duration*/
int state_duration(int state)
{
  int Duration = 0;
  switch (state)
  {
    case PLAY:
      Duration = (int)exponential(PLAY_Dura);
    case STOP:
      Duration = (int)exponential(STOP_Dura);
    case PAUSE:
      Duration = (int)exponential(PAUSE_Dura);
    case FF:
       Duration = (int)exponential(FFRW_Dura);
    case RW:
       Duration = (int)exponential(FFRW_Dura);
    case SM:
       Duration = (int)exponential(SM_Dura);
    case JUMP_F:
       Duration = (int)exponential(JUMP_Dura);
    case JUMP_B:
```

```
        Duration = (int)exponential(JUMP_Dura);
  }
  Duration = (int)(Duration * DURATION_FACTOR);
  if (Duration <= 0) return 1;
  else return Duration;
}


/* randomly return a jump point base on current playpoint and jump direction.
   According to the state machine, previous state before jump must be PLAY. */
int pick_jump_point(struct Client *C, int jump_direction)
{
  //offset means current offset
  int offset = C->offset + (int)simtime() - C->update_time; //assume has been in play
state since last update_time
  double prob = prob();
  if (jump_direction == JUMP_B)
  {/*return a playpoint before C's current offset*/
    return (int)(offset * prob);
  }
  else if (jump_direction == JUMP_F)
  {/*return a playpoint after C's current offset*/
    return (int)(offset + (m[C->movieID].duration*60 - offset) * prob);
  }
  else
  { /*it should never come into this branch. added it here just in case*/
    return offset;
  }
}

/*check whether given client is supported by local buffer only. Return TRUE or FALSE
   Given client's state is supposed to be PLAY, PAUSE(? must revisit to confirm) or
PIC_VCR */
int on_local_buffer_only(struct Client *C)
{
  if(C->state == STOP || C->state == JUMP_B || C->state == JUMP_F)
  {
    printf("Function: on_local_buffer_only should not be called for state %d \n", C-
>state);
    return FALSE;
  }
  if(C->current_stream == NULL)
  {
    return TRUE;
  }
  else
  {
    return FALSE;
  }
}

/*check whether local buffer can support client's Pic VCR request. Return TRUE or FALSE
  so far only one call used: void pic_VCR(struct Client *C, int new_state).
  Current implementation is based on estimation only because this simulation does record
  local buffer */
int check_local_buffer(struct Client *C, double VCR_speed)
{
      int playpoint = C->offset + (int)simtime() - C->update_time;
      int stream_playpoint;
    if(VCR_speed > 1) //FF
    {
      if(C->current_stream != NULL)
      {
       stream_playpoint = C->current_stream->offset + (int)simtime() - C->current_stream-
>launchtime;
       if(playpoint < stream_playpoint)
       {
               return TRUE;
       }
       else
       {
               return FALSE;
       }
      }
      else //C->current_stream == NULL
      {
       return FALSE;
      }
    }
    else if(VCR_speed > 0) //slow motion forward or play
    {
```

```
        if(C->current_stream != NULL)
       {
        return TRUE;
       }
       else
       {
        return FALSE;
       }
    }
    else //RW or SM backward
    {
        return TRUE;
    }
}

/* play()function calls to check whether a playpoint is in client's buffer and the
new_duration can be
entirely supported on local buffer; return TRUE or FALSE*/
int in_buffer(struct Client *C, int playpoint)
{
  int C_latest_playpoint;
  int C_1st_page_DLed_fr_crrnt_strm_prnt;
  int stream_playpoint;
  int stream_parent_playpoint;
  switch (C->state)  //C->state is the previous state, future state should be PLAY
  {//only need to take care of the cases from which will call this function
    case JUMP_B:
      C_latest_playpoint = C->offset + (int)simtime() - C->update_time;
        if(C->offset <= playpoint)
      {//C's state before jump must be PLAY, C->offset is still at the point when that
PLAY started
       //hence as long as the backup jump is not earlier than the offset it should be in
buffer unless the
           //buffer size limited it.
       if(C->current_stream != NULL)
       {
               stream_playpoint = C->current_stream->offset + (int)simtime() - C-
>current_stream->launchtime;
               if(C->current_stream->parent == NULL)
               {
                       if((int)simtime() - C->update_time <= CLIENT_BUFFER -
(stream_playpoint - C_latest_playpoint) )
                       {//time C played is less or equal to the buffer that can be used,
which is the total buffer size
                        //minus buffer used for saving incoming frames
                               return TRUE;
                       }
                       else
                       {//C->offset is already overwritten in buffer. Thus the earliest
frame can be found is
                        //entirely up to buffer size used for frames that have been played
                               if( playpoint >= C_latest_playpoint - (CLIENT_BUFFER -
(stream_playpoint - C_latest_playpoint)))
                               {//if the jumppoint is later than the earliest frame saved
                                       return TRUE;
                               }
                               else
                               {
                                       return FALSE;
                               }
                       }
               }
               else //C->current_stream has parent, which means another chunk of buffer
is used for incoming videos
               {    //from C->current_stream->parent
                       stream_parent_playpoint = C->current_stream->parent->offset +
(int)simtime() - C->current_stream->parent->launchtime; //wrong?
                       if((int)simtime() - C->update_time <= CLIENT_BUFFER -
(stream_playpoint - C_latest_playpoint)
                                                                                   -
(stream_parent_playpoint - stream_playpoint) )
                       {//time C played is less or equal to the buffer that can be used,
which is the total buffer size
                        //minus buffer used for saving incoming frames
                               return TRUE;
                       }
                       else
                       {//C->offset is already overwritten in buffer. Thus the earliest
frame can be found is
                        //entirely up to buffer size used for frames that have been played
```

```
                              if( playpoint >= C_latest_playpoint - (CLIENT_BUFFER -
(stream_playpoint - C_latest_playpoint)
                                      - (stream_parent_playpoint - stream_playpoint)))
                              {//if the jumppoint is later than the earliest frame saved
                                      return TRUE;
                              }
                              else
                              {
                                      return FALSE;
                              }
                      }
              }
       }
       else //C->current_stream == NULL
       {
              if((int)simtime() - C->update_time <= CLIENT_BUFFER)
              {// current time minus C->update_time, which is the beginning of the last
PLAY state, is smaller
               // than the buffer size
                      return TRUE;
              }
              else if ( 0 <= C->offset + C->duration - CLIENT_BUFFER <= playpoint)
              {// the jump point is still later than the earliest frame saved in the
buffer
                      return TRUE;
              }
              else // shouldn't be here: buffer cannot support the play longer than
buffer size.
              {//add here just in case
                      return FALSE;
              }
       }
      }
      else //C->offset > playpoint
      {
       return FALSE;
      }
    case JUMP_F:
       if(C->current_stream != NULL)
      {
       stream_playpoint = C->current_stream->offset + (int)simtime() - C->current_stream-
>launchtime;
       if(playpoint <= stream_playpoint)
       {//jumppoint is earlier than current_stream's current playpoint. Jumppoint is
later than C's latest playpoint by given
        //because it's JUMP_F. Hence the jumppoint must be in buffer
              return TRUE;
       }
       else //playpoint > stream_playpoint, which means jumppoint is later than
current_stream's current playpoint
       {
              if(C->current_stream->parent != NULL)
              {
                      stream_parent_playpoint = C->current_stream->parent->offset +
(int)simtime() - C->current_stream->parent->launchtime;
                      C_1st_page_DLed_fr_crrnt_strm_prnt = C->current_stream->parent-
>offset +
                                                              (C->current_stream->launchtime
+ C->current_stream->lifetime) - //time current stream merges into its parent

C->current_stream->parent->launchtime - //total value now is the playpoint of current
stream's parent's when current stream merges

(stream_parent_playpoint - stream_playpoint); //minus the gap of current stream and its
parent's playpoints
                      if(C_1st_page_DLed_fr_crrnt_strm_prnt <= playpoint <=
stream_parent_playpoint)
                      //playpoint after jump is earlier than current_stream's parent's
playpoint and later than first frame
                      //client downloaded from current_stream's parent. Hence the
playpoint after jump must be in buffer
                      {
                              return TRUE;
                      }
                      else
                      {
                              return FALSE;
                      }
              }
              else //C->current_stream->parent == NULL
              {
```

```
                     return FALSE;
              }
       }
      }
      else //C->current_stream == NULL
      {
       if(playpoint < C->offset + C->duration)
       {//C->offset + C->duration is the latest possible known frame in buffer
              return TRUE;
       }
       else
       {
              return FALSE;
       }
      }
    case PAUSE:
        return TRUE;
    case FF:
        return FALSE;
    case RW:
        return TRUE;
      case SM:
      if(C->current_stream != NULL)
      {
       return TRUE;
      }
      else
      {//C->current_stream == NULL
       if(C->speed > 0)
       {
              return FALSE;
       }
       else //C->speed must be < 0
       {
              return TRUE;
       }
      }
    case PLAY:
        return TRUE;
    case STOP:
        return FALSE;
      default:
       return FALSE;
  }
}

/*sub-function of void play(struct Client *C)*/
void play_from_jump(struct Client *C, int new_duration)
{
    double new_speed = VCR_speed(PLAY);
    Stream *temp;
    int resume_point = pick_jump_point(C, C->state);
    // randomly find a jump point base on playpoint before jump and probability chart
    if (in_buffer(C, resume_point) == TRUE)
    {//found resume point in local buffer, do nothing but update client info
        if(C->current_stream != NULL && C->current_stream->parent != NULL && resume_point
> C->offset)
        { //client refers to a stream and the stream has parent (means the stream is
still catching up with its parent),
          //then jump forward and found resume point in local buffer means this client
can refer to current stream's
          //parent instead of current stream
            temp = C->current_stream->parent;
            temp->clientCt++; //in case parent's clientCt is 1
            leave_current_stream(C);  // leave current stream
            temp->clientCt--; //to reflect the truth
            join_stream(C, temp);
        }
        update_client_state(C, PLAY, resume_point, new_speed, new_duration);
    }
    else //not found resume point in local buffer
    {
        if (Admission() == TRUE || (C->current_stream->clientCt== 1 && C->current_stream-
>first_kid == NULL
            && C->current_stream->root == FALSE))
        {// passed admission or C is the only client referring to C->current_stream
            leave_current_stream(C); //leave current stream if have one.
            //if Admission() was FALSE, it should return TRUE now
            //since C->current_stream must already be relieved
            temp = New_stream(C->movieID, resume_point);
            //temp->type = INTERACTIVE;
```

```
            stream_Session(temp);
            if (temp != NULL)
            {//since admission passed, this case is guaranteed, but keep the check here
just in case
                join_stream(C, temp);
                update_client_state(C, PLAY, resume_point, new_speed, new_duration);
            }
            else
            {
                C->state = PLAY;
            }
        }//end if
        else // cannot pass admission
        {//go back to original play status before jump
            C->state = PLAY; //this is the only necessary change to C since C->state from
this line since the only
          //thing changed when C's state became jump
          // C->offset no change since it's still the old one when state was PLAY
          // C->update_time no change since it's still the old one when state was PLAY
         // rejection_VCR++; //terminate request
        }//end of else
    }//end of else
}
/*VCR operation, sub-function of check_VCR(Client *C) */
void play(struct Client *C)
{
  int resume_point;
  double new_speed = VCR_speed(PLAY);
  int new_duration = state_duration(PLAY);
  Stream *temp;
  switch (C->state) //previous state
  {
    case JUMP_B:  //resume from jump backward
    {
      play_from_jump(C, new_duration);
      break;
    }

    case JUMP_F:  //resume from jump forward
    {
      play_from_jump(C, new_duration);
      break;
    }

    case PAUSE: //resume from pause
    {
      /* 1. if client is still referring to any stream; can continue refer to it
          2. if client is no longer referring to any stream, local buffer local buffer is
supposed to be full. So it can
          support for CLIENT_BUFFER long playback time on local buffer only.
          In either case above, no need to do anything but update client's information */
      //offset's value doesn't matter since the update function doesn't really need it
      update_client_state(C, PLAY, 0, new_speed, new_duration);
      break;
    }

    case STOP:
    {//this case is impossible since STOP state is not able to go to any other state
      printf("There must be something wrong since STOP state is not able to go to any
other state such as PLAY \n");
      break;
    }

    case PLAY:
    {//this case means no change of VCR state because it's from "PLAY" to "PLAY"
      //offset's value doesn't matter since the update function doesn't really need it
      resume_point = find_playpoint(C);
      if(C->current_stream != NULL)
        {
         update_client_state(C, PLAY, resume_point, new_speed, new_duration);
        }
        else
        {
         if (Admission() == TRUE)
         {//no current stream or current stream is not isolated; but passed admission
          temp = New_stream(C->movieID, resume_point);
               //temp->type = INTERACTIVE;
          stream_Session(temp);
          if (temp != NULL)
          {//since admission passed, this case is guaranteed, but keep this check here
just in case
```

```
                join_stream(C, temp);
                update_client_state(C, PLAY, resume_point, new_speed, new_duration);
        }
        else //since admission passed, this case is impossible, but keep it here just in
case
        { //reject PLAY request
                stop(C);
                //  rejection_VCR++; //terminate request
        }
       }//end of if
       else //cannot pass admission
       {// reject PLAY request
        stop(C);
        //rejection_VCR++; //terminate request
       }//end of else
      }
      break;
    }

    default: //resume from VCR with picture
    {
      resume_point = find_playpoint(C);
      if(resume_point < 0 || resume_point >= m[C->movieID].duration*60)
      {
        stop(C);
        break;
      }
      if (in_buffer(C, resume_point) == TRUE)
      { //found resume point in local buffer
        update_client_state(C, PLAY, resume_point, new_speed, new_duration);
      }
      else //not found resume point in local buffer
      {
         if (C->current_stream != NULL && C->current_stream->root == FALSE &&
          C->current_stream->parent == NULL && C->current_stream->clientCt == 1 &&
          C->current_stream->first_kid == NULL)
        {//if client has an isolated stream, which was used to support PIC_VCR; now use
it for PLAY patching support
          if(C->current_stream->C_head != C || C->current_stream->C_tail != C)
          {
              int time_now = (int)simtime();
         printf("current time is %d, stream %X \n", time_now, C->current_stream);
         printf("Isolated stream is not really isolated\n");
          }
          C->current_stream->movieID = C->movieID;
          C->current_stream->launchtime = (int)simtime();
          C->current_stream->offset = resume_point;
          C->current_stream->starttime = C->current_stream->launchtime - C-
>current_stream->offset;
          C->current_stream->clientCt = 1;
          C->current_stream->root = FALSE;
          C->current_stream->parent = Parent(C->current_stream);
          if (C->current_stream->parent == NULL)
          { // no parent found for C's current_stream; then C's current_stream becomes
root, and being added in to multicast stream list of movieID
            C->current_stream->root = TRUE;
            C->current_stream->type = MULTICAST;
            C->current_stream->older_sibling = NULL;
            C->current_stream->younger_sibling=NULL;
            C->current_stream->first_kid = NULL;
            C->current_stream->lifetime = m[C->current_stream->movieID].duration*60 -
resume_point;
            insert_multicast_stream(C->current_stream);
          }
          else
          { // found parent for C's new current_stream
            if (C->current_stream->parent->root == TRUE) C->current_stream->type = MERGE;
            else C->current_stream->type = ADMISSION;
            C->current_stream->first_kid = NULL;
            insert_non_multicast_stream(C->current_stream);
            //update_stream(C->current_stream->parent);

            /*if (C->current_stream->starttime == C->current_stream->parent->starttime)
            {
                printf("whoho!, there it is. the special case C->current_stream starttime
== C->current_stream->parent's, C->current_stream = %X\n", C->current_stream);
            }*/
          }//end else
          update_client_state(C, PLAY, resume_point, new_speed, new_duration);
        }//end if
```

```
        else if (Admission() == TRUE || (C->current_stream != NULL && C->current_stream->clientCt== 1
            && C->current_stream->first_kid == NULL && C->current_stream->root == FALSE))
        {//no current stream or current stream is not isolated; but passed admission or C is the only client referring to C->current_stream
          leave_current_stream(C); //leave current stream if have one.
          //if Admission() was FALSE, it should return TRUE now
          //since C->current_stream must already be relieved
          temp = New_stream(C->movieID, resume_point);
          //temp->type = INTERACTIVE;
          stream_Session(temp);
          if (temp != NULL)
          {//since admission passed, this case is guaranteed, but keep this check here just in case
            join_stream(C, temp);
            update_client_state(C, PLAY, resume_point, new_speed, new_duration);
          }
          else //since admission passed, this case is impossible, but keep it here just in case
          { //reject PLAY request
            stop(C);
          //  rejection_VCR++; //terminate request
          }
        }//end of else if
        else //no current stream or current stream is in a tree; and cannot pass admission
        {// reject PLAY request
          stop(C);
          //rejection_VCR++; //terminate request
        }//end of else
      }//end of else
      break;
    }//end of PIC_VCR case
  }//end of switch
}


/*VCR operation, sub-function of check_VCR(Client *C)
   C->state before entering this function must be PLAY.
   The client which C points to should be removed after
   this stop  */
void stop(struct Client *C)
{ //leave current stream
  int new_state = STOP;
  double new_speed = VCR_speed(new_state); //dummy speed
  int new_duration = state_duration(new_state);
  int new_offset = 0; //dummy offset
  //only jump really needs a new offset, other state's new_offset is just a dummy argument because update_client_state won't new it
  leave_current_stream(C);
  update_client_state(C, new_state, new_offset, new_speed, new_duration);
}

/*VCR operation, sub-function of check_VCR(Client *C)
   C->state before entering this function must be PLAY */
void pause(struct Client *C)
{ //do not leave current stream yet until local buffer is full, will check in function check_VCR */
  int new_state = PAUSE;
  double new_speed = VCR_speed(new_state); //dummy speed
  int new_duration = state_duration(new_state);
  int new_offset = 0; //dummy offset
  //only jump really needs a new offset, other state's new_offset is just a dummy argument because update_client_state won't new it
  update_client_state(C, new_state, new_offset, new_speed, new_duration);
}

/*VCR operation, sub-function of check_VCR(Client *C)
   C->state before entering this function must be PLAY */
void jump(struct Client *C, int new_state)
{
  /*do not leave current stream yet until the client has to, will check when resume */
  //find new speed, new duration and jump point
  /*double new_speed = VCR_speed(new_state); //dummy speed
  int new_duration = state_duration(new_state);
  int new_offset = 0; //dummy offset
  if(new_state == JUMP_B || new_state == JUMP_F)
  {//only jump really needs a new offset
      new_offset = pick_jump_point(C, new_state);
  }
```

```
  //only jump really needs a new offset, other state's new_offset is just a dummy
argument because update_client_state won't new it
  update_client_state(C, new_state, new_offset, new_speed, new_duration);*/
  C->state = new_state;
}

/*VCR operation, sub-function of check_VCR(Client *C)
   C->state before entering this function must be PLAY */
void pic_VCR(struct Client *C, int new_state)
{
//find new speed, new duration and jump point(for jump state only)
  double new_speed = VCR_speed(new_state); //real speed
  int new_duration = state_duration(new_state);
  int new_offset = 0; //dummy offset
  Stream *temp;
  //only jump really needs a new offset, other state's new_offset is just a dummy
argument because update_client_state won't new it

  if(check_local_buffer(C, new_speed) == TRUE)
  {/*local buffer can support; so do not leave current stream yet until the client has to;
      will check in function check_VCR*/
    update_client_state(C, new_state, new_offset, new_speed, new_duration);
  }
  else if (Admission() == TRUE ||
  (C->current_stream != NULL && C->current_stream->clientCt == 1
  && C->current_stream->first_kid == NULL && C->current_stream->root == FALSE) )
  {/* local buffer cannot support; but got admission */
    /* allocate an isolated stream to support PIC_VCR*/
    leave_current_stream(C); /*leave current stream if have one. */
    /*if Admission() was FALSE, it should return TRUE now
       since C->current_stream must already be relieved */
    update_client_state(C, new_state, new_offset, new_speed, new_duration);
    /*get an isolated stream to support current VCR+Picture operation*/
    temp = new_isolated_stream(C);
    join_stream(C, temp);
  }
  else /* cannot pass admission */
  {
     /*reject VCR request, client keeps on PLAY mode*/
     //should be the same as before, added just in case.
      new_duration = state_duration(PLAY);
      update_client_state(C, PLAY, new_offset, PLAY_SPEED, new_duration);
    //rejection_VCR++;
  }
}

/*check if the end of movie is reached while playing or VCR+PIC; return TRUE or FALSE*/
int check_stop(struct Client *C)
{
  if(C->state == JUMP_B || C->state == JUMP_F || C->state == PAUSE)
  {
    return FALSE;
  }
  else if (C->state == STOP)
  {
    return TRUE;
  }
  else if (C->state == PLAY)
  {
    if (C->offset + (int)simtime() - C->update_time >= m[C->movieID].duration*60)
    {
      return TRUE;
    }
    else
    {
      return FALSE;
    }
  }
  else //Pic_VCR: FF, RW, SM
  {
    if ((int)(C->offset + ((int)simtime() - C->update_time)*C->speed) >= m[C-
>movieID].duration*60
        || (int)(C->offset + ((int)simtime() - C->update_time)*C->speed) <= 0 )
    {
      return TRUE;
    }
    else
    {
      return FALSE;
    }
  }
```

```
}

/* Only called by function void maintain_VCR(struct Client *C)
played or VCR+PIC on buffer (if state is PIC_VCR, client might or might not still refer
to a stream;
if state is PLAY and client still refers to a stream, there is no need to call this
function; if PLAY on
buffer only, there is need) . This function checks whether the buffer still is able to
support;
return TRUE or FALSE*/
int end_buffer_support(struct Client *C)
{
      int played_time;
      if (C->speed > 0)
      {
       played_time = (int)((simtime() - C->update_time) * C->speed);
      }
      else
      {
       played_time = (int)((simtime() - C->update_time) * C->speed * (-1));
      }

      if( played_time >= exponential(CLIENT_BUFFER))
      {
       return TRUE;
      }
      else
      {
       return FALSE;
      }
}

/*only for PlAY and VCR_PIC, return current playpoint base on client's previous playpoint,
current time and VCR speed*/
int find_playpoint(struct Client *C)
{
  int offset;
  if(C->state == JUMP_B || C->state == JUMP_F || C->state == PAUSE)
  {
    return C->offset;
  }
  else if (C->state == STOP)
  {
    return m[C->movieID].duration*60;
  }
  else if (C->state == PLAY)
  {
    offset = C->offset + (int)simtime() - C->update_time;
    return offset;
  }
  else //Pic_VCR: FF, RW, SM
  {
    offset = (int)(C->offset + ((int)simtime() - C->update_time)*C->speed);
    return offset;
  }
}

/*check if given client's local buffer is full during PAUSE or PIC_STATE state and
downloading from referred stream;
   return TRUE or FALSE
  so far only called by void maintain_VCR(struct Client *C) */
int local_buffer_full(struct Client *C)
{
  if (C->state == PAUSE || C->state == FF || C->state == RW || C->state == SM)
  {//because no matter what current state is, the download speed is fixed as play speed,
which is 1
    if ((int)simtime() - C->update_time >= CLIENT_BUFFER )
    {
      return TRUE;
    }
    else
    {
      return FALSE;
    }
  }
  else
  {//not suppose to choose this branch at all
      return FALSE;
  }
}
```

```
/* given client leaves the stream that was referred*/
void leave_current_stream(struct Client *C)
{
      if (C->current_stream != NULL) // client C has a stream before VCR operation
      {
       release_one_client(C);
      }
}

/* check whether a given client requests a VCR operation and handle it*/
void check_VCR(struct Client *C)
{
  int new_state = VCR_pick(C->state);  /*randomly pick a VCR operation base on current
state*/
  switch (new_state)
  {
    case PLAY: /*play or resume*/
    {
      play(C);
      break;
    }

    /*stop case here can only be triggered if client wants to stop play before the end
      If its probability is 0 in the chart; VCR_pick() will never pick "STOP", which
      means "STOP" can only be reached naturally when movie ends. Natural stop case
      is checked and handled after "else if"*/
    case STOP:
    {
      stop(C);
      break;
    }

    /* VCR actions w/o pictures: PAUSE and JUMP */

    /*PAUSE, no need to remove the client, no need to drop the current stream right away.
      Time stay in PAUSE varies*/
    case PAUSE:
    {
      pause(C);
      break;
    }

    /*Difference between JUMP and PAUSE is
      1. Time stay in JUMP is always only 1 time because VCR_pick() will pick PLAY when
         input state is JUMP
      2. Afer PAUSE, resume point is fixed; after JUMP, resume point varies*/
    case JUMP_B:
    {
      jump(C, new_state);
      break;
    }

    case JUMP_F:
    {
      jump(C, new_state);
      break;
    }

    /* VCR actions w/ pictures */
    case FF:
    {
      pic_VCR(C, new_state);
      break;
    }

    case RW:
    {
      pic_VCR(C, new_state);
      break;
    }

    case SM:
    {
      pic_VCR(C, new_state);
      break;
    }
  }//end of switch
}//end of check_VCR

void maintain_VCR(struct Client *C)
{
```

```
  int playpoint;
  Stream *temp;
  if(C->state == JUMP_F || C->state == JUMP_B)
  {//not suppose to come into this branch because Jump is a state with duration 1. It
never lasts
   //put it here just in case
    return;
  }
  if (C->state == PAUSE)
  {
    if (on_local_buffer_only(C)== FALSE)
    { /*paused but still using current stream to fill out local buffer*/
      if (local_buffer_full(C) == TRUE)
      { /* stop referring to the stream since local buffer is full*/
        leave_current_stream(C);
      }
    }
  }
  else if (C->state == STOP)
  {/*stop case should not be triggered if client's previous state was STOP.
        put here just in case*/
    stop(C);
  }
  else if (check_stop(C) == TRUE)
  {/*state should be PLAY or PIC_VCR, check if the end of movie is reached naturally*/
    stop(C);
  }
  else if (on_local_buffer_only(C)== TRUE && end_buffer_support(C)== TRUE)
  {/*played on buffer only but buffer no longer supports*/
    if (Admission() == TRUE) /*admission control needed to implement*/
    {
      if (C->state == PLAY)
      {
        playpoint = find_playpoint(C);
        if(playpoint >= m[C->movieID].duration*60) stop(C);
        else
        {
          temp = New_stream(C->movieID, playpoint);
          stream_Session(temp);
          join_stream(C, temp);
        }
      }
      else  /*C->state == PIC_VCR*/
      {
        playpoint = find_playpoint(C);
        if(playpoint < 0 || playpoint >= m[C->movieID].duration*60) stop(C);
        else
        {
          temp = new_isolated_stream(C);
          join_stream(C, temp);
        }
      }
    }
    else
    { /*cannot pass admission*/
      stop(C);
     // rejection_VCR++; /*terminate request*/
    }//end of else
  }//end of else if
  else if ((C->state == FF || C->state == RW || C->state == SM) &&
on_local_buffer_only(C)== FALSE &&
              (C->current_stream->root == TRUE || C->current_stream->parent != NULL) )
  {//state is VCR with picture mode, and not on local buffer only
   //playing on buffer but still downloading from stream which is not an isolated stream
    if (end_buffer_support(C)== TRUE )
    {//buffer can no longer support current VCR operation
      if (Admission() == TRUE ||
          (C->current_stream != NULL && C->current_stream->clientCt == 1
          && C->current_stream->first_kid == NULL && C->current_stream->root == FALSE))
      { //passed admission or C is the only client referring to C->current_stream
        leave_current_stream(C); /*leave current stream if have one. */
        /*if Admission() was FALSE, it should return TRUE now
           since C->current_stream must already be relieved */
        /*get an isolated stream to support current VCR+Picture operation*/
        playpoint = find_playpoint(C);
        if(playpoint < 0 || playpoint >= m[C->movieID].duration*60) stop(C);
        else
        {
          temp = new_isolated_stream(C);
          join_stream(C, temp);
        }
```

```
      }
      else
      { //cannot pass admission
        stop(C);
        //rejection_VCR++; //terminate request
      }
    }
    else if (local_buffer_full(C) == TRUE)
    { /* local buffer can still support current VCR operation;
         But stop referring to the stream since local buffer is full*/
      leave_current_stream(C);
    }
  }//end of else if
}

/* allocate a new stream for a movie and a play point without looking for a parent*/
struct Stream * new_isolated_stream(struct Client *C)
{
  Stream* PS;
  //channel_capacity--;
  PS = malloc(sizeof *PS);
  //channel_capacity--;
  PS->movieID = C->movieID;
  PS->offset = find_playpoint(C);
  PS->launchtime = (int)simtime();
  PS->starttime = PS->launchtime - PS->offset;
  PS->clientCt = 0;
  PS->first_kid = NULL;
  PS->parent = NULL;
  PS->root = FALSE;
  PS->older_sibling= NULL;
  PS->younger_sibling=NULL;
  PS->time_last_kid_merges = 0; //the value 0 is meaningless, just to initialize it here
  PS->ended = FALSE;
  PS->C_head = NULL;
  PS->C_tail = NULL;
  PS->type = INTERACTIVE;
  if(C->state == FF || C->state == SM)
  {
    PS->lifetime = (int)((m[PS->movieID].duration*60 - PS->offset)/C->speed) + 2;
  }
  else if (C->state == RW)
  {
      PS->lifetime = (C->offset/abs((int)C->speed)) + 2;
  }
  stream_Session(PS);
  return PS;
}

/* zipf function from Dr.Kim */
void zipf_init()
{
  double C;
  int i;
  double k;
  double alpha = theta * -1.0;

  for(i=0;i < NUM_MOVIES;i++)
  {
    zipf[i] = 0.0;
  }

  C = 0.0;
  k = 1.0;
  for(i=1 ; i<=NUM_MOVIES ; i++)
  {
    C += pow(1/k, 1.0 + alpha)  ;
    k++;
  } /* for */

  C = 1.0/C;
  k = 1.0;
  for(i=1 ; i <=NUM_MOVIES ; i++)
  {
    zipf[i] = zipf[i-1] + C/pow(k,1.0 + alpha);
    k++;
  } /* for */
} /* zipf_init */

/* CDF function to choose movies using zipf function */
int choose_movie()
```

```
{
  double ran_num;
  int i = 1;
  ran_num = prob();
  while(ran_num > zipf[i]) i++;
  return(i);
} /* choose_movie */


//now some verification functions
void result_print(int val)
{
  printf(" %d\n", val);
}

/*add results after each run*/
void add_results(int val, double *my_var)
{
  *my_var =  *my_var + val;
}
```