Handling Updates and Crashes in VoD Systems *

Eenjun Hwang, **Kemal Kilic** and **V.S. Subrahmanian**

Computer Science Department
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20770
{hwang, kemal, vs}@cs.umd.edu

Abstract

Though there have been several recent efforts to develop disk based video servers, these approaches have all ignored the topic of updates and disk server crashes. In this paper, we present a *priority based model* for building video servers that handle two classes of events: *user* events that could include `enter, play, pause, rewind, fast-forward, exit`, as well as *system* events such as `insert, delete, server-down, server-up` that correspond to uploading new movie blocks onto the disk(s), eliminating existing blocks from the disk(s), and/or experiencing a disk server crash. We will present algorithms to handle such events. Our algorithms are provably correct, and computable in polynomial time. Furthermore, we guarantee that under certain reasonable conditions, continuing clients experience jitter free presentations. We further justify the efficiency of our techniques with a prototype implementation and experimental results.

*This work was supported by the Army Research Office under Grant Nr. DAAH-04-95-10174, by the Air Force Office of Scientific Research under Grant Nr. F49620-93-1-0065, by ARPA/Rome Labs contract F30602-93-C-0241 (ARPA Order Nr. A716), by Army Research Laboratory under Cooperative Agreement DAAL01-96-2-0002 Federated Laboratory ATIRP Consortium and by an NSF Young Investigator award IRI-93-57756.

1 Introduction

Over the last few years, there has been a tremendous drop in digitization costs, accompanied by a concomitant drop in prices of secondary and tertiary storage facilities, and advances in sophisticated compression technology. These three advances, amongst others, have caused a great increase in the quality and quantity of research into the design of video servers [2, 4, 11, 6, 10, 16].

Most models of video servers to date assume the following parameters:

1. Movies are stored, in part, or in their entirety, on one or more disks.
2. The video-on-demand VoD system is responsible for handling "events" that occur. Client events that have been studied include:
    - the `enter` of a new client into the system, requesting a movie,
    - the `exit` of an existing client from the system.
    - the activities of continuing clients (e.g. `play, fast forward, rewind, pause`).

    "Handling" an event refers to the process by which the VoD server assigns jobs to different disk servers, so as to optimize some performance criterion. A variety of algorithms to "handle" the above events have been studied by researchers.

All the above events are "user" events, in the sense that they are invoked or caused by the activities of a user of the VoD system. However, in reality, there is another class of events that must be accounted for, which we call *system events*, which includes events such as `server-down` (specifying that a certain disk server has crashed), `server-up` (specifying that a disk server that had previously crashed is "up" again), `insert` (specifying that the system manager wishes to include some new movies (or blocks of movies) on a disk, and `delete` (specifying that the system manager wishes to delete some movies from a server's disk array). Most work to date on server crashes has focused on the important topic of *recovery* of data on the crashed disk, but has not really looked into how to satisfy clients in the VoD system who were promised service based, in part, on the expectation that the crashed disk would satisfy some requests. *The main focus of this paper is to develop* VoD *server algorithms that can handle not just user events, but can also handle system events.*

The problem of *updates* in video servers is crucial for several applications where video data is being gathered at regular intervals and being placed on the VoD system. For example, a movie-on-demand vendor may, at regular intervals, include new movies in the repertoire of movies offered to potential customers. These movies need to be placed on the disk array that the vendor may be using, leading to an `insert` operation. Similarly, in news-on-demand systems, new news videos and audio reports may become available on a continuing basis, and these need to be made available to editors of news programs for creating their current and up to date news shows. In many similar systems today, this is done by taking the system "down", accomplishing the update, and then bringing the system back "up" again. The obvious undesirable aspect of this way of handling updates is that service must be denied to customers who wish to access the server when it is down, thus leading to lost revenues for the VoD vendor. The algorithms proposed in this paper treat updates as (collections of) events, and schedule them to occur concurrently with user-events in a manner that ensures that:

1. existing customers see no deterioration (under some reasonable restrictions) in the quality of service, and
2. the update gets incorporated in a timely fashion. In particular, our algorithms will flexibly adapt to the load on the disks, so as to incorporate as much of the update as possible when resources are available, and to reduce the update rate when resources have been previously committed.
3. the system is not "taken down" in order to accomplish the update.

Unlike the issue of updates, disk crashes have certainly been studied extensively over the years [2, 12]. However, consider the problem of a VoD server that has made certain commitments to customers. When a crash occurs, the VoD server must try to ensure that any client being serviced by the disk that crashed be "switched" to another disk that can service that client's needs. *Furthermore, the VoD server must ensure that the fact disk d has crashed be taken into account when processing new events.* In the same vein, when a disk server that had previously crashed comes back "up", this means that new system resources are available, thus enabling the VoD server to take appropriate actions (e.g. admit waiting clients, re-distribute the load on servers to achieve good load balance, etc.). *We show how our framework for handling updates can handle such crashes as well (under certain limitations of course).*

In particular, we propose an algorithm called the VSUC ("Video Server with Updates and Crashes") algorithm, that handles events (including user-initiated events, as well as update events and crashes) and has several nice properties. In particular:

- VSUC guarantees that under certain conditions, it ensures continuous, jitter free service for clients, once they have been admitted. (We will make the conditions precise in Theorem 4.1).
- VSUC also guarantees (again under certain conditions), that no client is denied service for arbitrarily long (cf. Theorem 4.2).
- VSUC reacts to client initiated, as well as system (update/crash) events in polynomial time.

2 System Architecture

Throughout this paper, we will use the term *video block* (or just *block*) to denote a video segment. We will assume that the size of a block is arbitrary, but fixed. In other words, one VoD application may choose a block to be of size 30 frames, while another may consider it to be of size 60 frames. As our video data is stored on disk, this means that the start of each *video block* is located on a single page of any disk that contains the block.

As data is laid out on a collection of disks, we will assume that this collection of disks is partitioned into disjoint subsets $DC_1, \ldots, DC_n$. We will furthermore assume that all disks in DC_i are homogeneous (i.e. have identical characteristics) and a single disk server DS_i regulates access to the disk drives in DC_i. It is entirely possible that DC_i contains only one disk, but it could contain

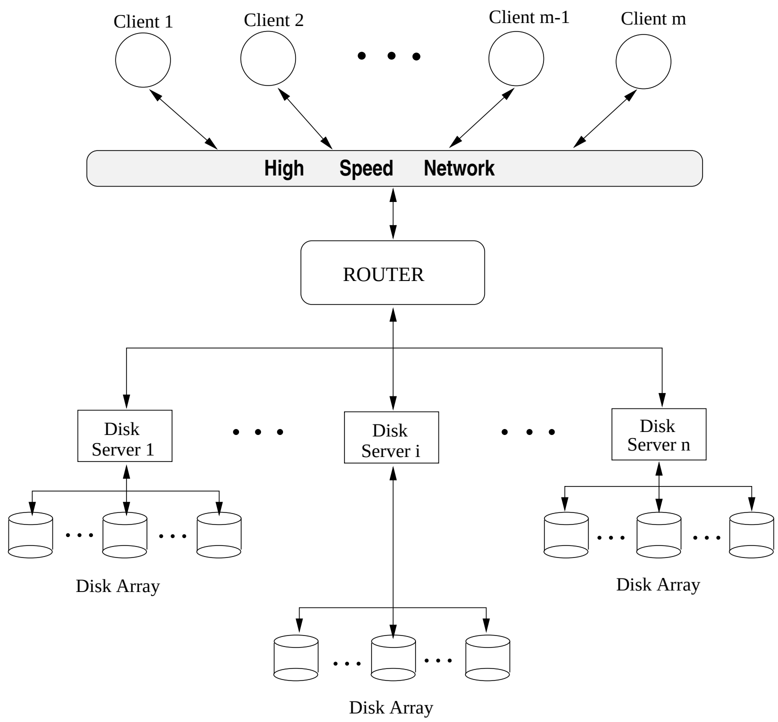


Figure 1: System structure.

more. *Note that there is no requirement that two disk collections DC_i, DC_j need to have the same characteristics* and hence, disks in DC_i may have vastly different characteristics than those in DC_j – this is what accomplishes heterogeneity.

The design of disk servers is now well known [2, 4, 11, 6, 10, 16]. In its simplest form, a disk server is a piece of software that, given a physical disk address, retrieves the object located at that address. In our case, disk servers DS_i mediate access to a collection DC_i of disks, which means that given a `disk-id` and a physical disk address, the server retrieves the block located at the given disk address on the specified disk. Figure 1 shows the structure of the system as described informally above.

In our architecture, the video server is responsible for the following tasks:

1. When an event (user event or system event) occurs, the video server must determine how to handle the event. This may involve dispatching instructions to the disk servers. For example, such instructions could include: *Fetch (for client cl_id) the block starting on page p of disk d.* Note that the disk server does not necessarily need to know the client's identity, cl_id.

2. In addition, the video server may need to "switch" clients from one disk server to another. For example, client cl_id_1 may be being served by disk server DS_1. If a new client cl_id_{20} requests a movie (or block) that is only available through disk server DS_1 and if disk server DS_1 is already functioning at peak capacity, then it may be possible to "switch" client cl_id_1 to another disk server (say DS_2) if disk server DS_2 has the resources needed to satisfy client cl_id_1.

3. Third, the video server may "split" a job into smaller, manageable jobs, and distribute these smaller jobs to different servers, which leads to better system utilization.

4. Fourth, whenever an event (e.g. an update occurs), the video server must automatically create a schedule to accomplish the update, and determine what instructions must be sent to the disk server(s) involved to successfully handle the event.

5. Fifth, whenever events such as disk server crashes occur, the VoD server must re-assign the existing clients to other servers (when possible) and schedule *system generated recovery events* so as to minimize the damage caused by the crash.

2.1 System Parameters

In any VoD system, the participating entities may be divided into the following components:

1. Servers: these are the disk servers that retrieve specified blocks from the relevant disks;

2. Clients: these are the processes that are making/issuing requests to the servers; and

3. Data: this includes the movie data laid out on the disks.

In order to successfully model a VoD system, and develop provably correct and efficient algorithms for this purpose, we must model each of the above parameters, as well as the interactions between the above components.

Tables 2,3,4 show the notations we use to denote the relevant parameters of servers, clients, and movies, respectively.

Throughout this paper, we assume that there is a set $\mathcal{MOVIE} = \{\mathcal{M}_1, \ldots, \mathcal{M}_r\}$ of movies that we wish to store on disk. Each movie $\mathcal{M}_i$ has $\mathsf{bnum}(\mathcal{M}_i)$ "blocks". A block denotes the level of granularity at which we wish to store and reason about the media-data. For example, a block may be a single-frame (finest granularity) or a consecutive sequence of (100 frames). The application developer is free to select the size of a block in any way s/he wishes, but once such a block size is selected, s/he is committed to using the selected block size for the application. In other words, they are free to choose their block size as they wish, but once they make the choice, they must stick to it.

3 State Transition Model

In this paper, we will develop a *state transition* model that has the following properties:

- A *state* is any feasible configuration of the system, and includes information such as: which disk server(s) are serving a client, and what service they are providing the client, and what resources are committed by the server to the client to accomplish the service provided.

Symbol	Meaning
buf(i, s)	The total buffer space associated with the disk server i at state s.
cyctime(i, s)	The total cycle time for the server i at state s.
dtr(i, s)	The total disk bandwidth associated with the disk server i at state s.
timealloc(i, j, s)	The time-slice allocated to client j at state s by server i.
$\mu_s(i)$	The set of servers handling request by client i at state s.
$\wp(\mathcal{M}_i, b, s)$	The set of servers that contain block b of movie $\mathcal{M}_i$ according to placement mapping $\wp$ at state s.
d_active(i, s)	The set of all clients that have been assigned a non-zero time-allocation by disk server i at state s.
server_status(i, s)	The status flag for server i. It is true when the server is working, false otherwise.
switchtime(i, s)	The time required for the disk server i to switch from one client's job to another client's job at state s.
bufreq(i, j, s)	The buffer space needed at the server i to match the consumption rate of client j at state s.
priority(e, s)	The priority of the event e at state s.

Figure 2: Server Parameters

Symbol	Meaning
cons(i, s)	The consumption rate of client i at state s.
data(i, j, s)	The set of data blocks that server i is providing to client j at state s.
inuse(i, s)	This set consists of 3-tuples, $(j, \mathcal{M}_k, b)$, it specifies that the server i is providing block b of movie $\mathcal{M}_k$, to client j at state s.
active_client(s)	The set of all clients that are active at state s
m_active(s)	The set of all clients that are watching movie m at state s
rew_win(i, s)	The size of rewind window for client i at state s. This means the client can not rewind the movie more than that many blocks.
ff_win(i, s)	The size of fast forward window for client i at state s. This means the client can not fast forward the movie more than that many blocks.
watch_win$(\mathcal{M}_i, j, s)$	The time limit for client j to watch movie $\mathcal{M}_i$ at state s.
pause_win(i, s)	The time limit for client i to pause at state s.

Figure 3: Client Parameters

Symbol	Meaning
bnum$(\mathcal{M}_i)$	The number of blocks for a movie $\mathcal{M}_i$.

Figure 4: Movie Parameters

- The state of the system may change with time, and is triggered by *events*. Events include:
  - *Client-initiated events* such as `enter, exit, fast-forward, pause, rewind, play`, as well as
  - *Server-initiated events* such as `server-down, server-up` where a server goes "down" or comes back "up", and
  - *Manager-initiated events* such as `insert, delete`. Note that manager events could either be initiated by a human VoD system manager, or by a tertiary storage device that is staging data onto disk (though we will not go into this possibility in detail in this paper).

3.1 What is a State ?

A *system state* s consists of the following components:

1. A set `active_client`(s) of active clients at state s.
2. The current `cyctime`(i, s) of each server in the system.
3. The consumption rates of the active clients (`cons`(i, s)) in state s.
4. The time, `timealloc`(i, j, s), within `cyctime`(i, s) that has been allocated by server i to client j in state s.
5. The locations ($\wp(m, b, s)$) of each movie block, i.e. the set of all servers on which block b of movie m is located in state s.
6. The set of data blocks (`data`(i, j, s)) being provided by server i to client j in state s.
7. A client mapping μ_s which specifies, for each client C, a set of servers, $\mu_s(C)$, specifying which servers are serving client C.
8. A set `down_servers`(s) consisting of a set of servers that are down in state s.
9. A set `insert_list`(s) consisting of a set of 3-tuples of the form (i, m, b) where m is a movie, b is a block, and i is the server where this block will be inserted. (This set is used to model a set of insertion updates that are "yet to be handled.")
10. A set `delete_list`(s) consisting of a set of 3-tuples of the form (i, m, b) where m is a movie, b is a block, and i is the server where this block will be deleted. (This set is used to model a set of deletion updates that are "yet to be handled.") $\diamond$

For example, consider a very simple scenario where five movies $m_1, \ldots, m_5$ have been broken up into $60, 80, 50, 50, 60$ blocks each, and have been placed on the three disks as shown in Figure 5. Each disk hold some blocks of some movies, as indicated in the figure. An example of a state is the state s_5 shown below:

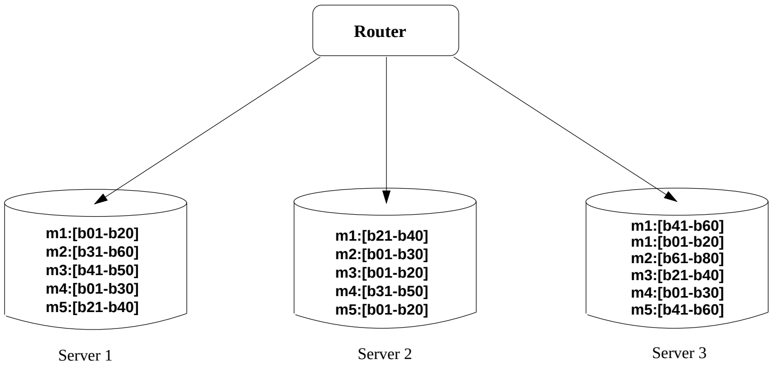


Figure 5: Example placement mapping

1. $\texttt{active\_client}(s_5) = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9\}$ indicates that 9 clients are currently being served.

2. $\texttt{cyctime}(i, s)$ may be 8, 10, 7 for the servers 1,2,3 respectively.

3. The consumption rates of the clients involved might be given by:

$$
\begin{aligned}
\texttt{cons}(c_1, s_5) &= 15. \\
\texttt{cons}(c_2, s_5) &= 25. \\
\texttt{cons}(c_3, s_5) &= 15. \\
\texttt{cons}(c_4, s_5) &= 10. \\
\texttt{cons}(c_5, s_5) &= 40. \\
\texttt{cons}(c_6, s_5) &= 10. \\
\texttt{cons}(c_7, s_5) &= 25. \\
\texttt{cons}(c_8, s_5) &= 15. \\
\texttt{cons}(c_9, s_5) &= 20.
\end{aligned}
$$

   These rates indicate, for example, that client c_1 is consuming data (e.g. outputting it on a display device) at a rate of 15 units/sec.

4. The following simple table may capture the assignment of time (within a cycle) for each client. Any client not explicitly assigned a time-slice by a server is assigned 0 time.

Server (i)	Client (j)	timealloc(i, j, s_5)
1	c_1	3
	c_4	2
	c_9	2
2	c_2	2.5
	c_5	2
	c_6	1
	c_8	1.5
	c_9	1
3	c_3	1.5
	c_5	2
	c_7	2.5

For example, the fact that server 1 does not explicitly list a time-allocation for client c_5 means that server 1 has assigned zero time for that client. Note that for each server i, it is the case that

$$\sum_{j=c_1}^{c_9} \texttt{timealloc}(i, j, s_5) \quad \leq \quad \texttt{cyctime}(i, s_5).$$

For example, the allocation for server 1 is given by:

$$3 + 2 + 2 = 7 \quad \leq \quad \texttt{cyctime}(1, s_5) = 8.$$

5. The data $\texttt{data}(i, j, s_5)$ being provided by server i to client j in state s_5 may, for example, be represented as the table:

Server (i)	Client (j)	data(i, j, s_5)
1	c_1	m1:[b2,b3]
	c_4	m2:[b30,b31]
	c_9	m4:[b4,b4]
2	c_2	idle
	c_5	m2:[b4,b4]
	c_6	m2:[b3,b3]
	c_8	m5:[b7,b7]
	c_9	m4:[b5,b5]
3	c_3	m2:[b67,b70]
	c_5	m2:[b5,b5]
	c_7	m3:[b5,b6]

6. Note that the preceding table fully specifies the client mapping. In this case, this mapping μ_{s_5} is given by:

$$\mu_{s_5}(c_1) = \mu_{s_5}(c_4) \quad = \quad \{1\}.$$

$$\begin{aligned}\mu_{s_5}(c_6) = \mu_{s_5}(c_8) &= \{2\}. \\ \mu_{s_5}(c_3) = \mu_{s_5}(c_7) &= \{3\}. \\ \mu_{s_5}(c_5) &= \{2,3\}. \\ \mu_{s_5}(c_9) &= \{1,2\}.\end{aligned}$$

Note that in the above table, some time has been allocated to clients that are idle, e.g. client c_2 has been allocated 2.5 time units by server 2, even though it is idle. This may be because this client has temporarily "paused" in which case, the reservations are still made for the client, but the disk is not actually reading data shipped by the server.

7. In this example, $\texttt{down\_servers}(s_5) = \emptyset$.

A system state s must satisfy certain simple constraints, that we list below.

1. *For each server i that is not down, the sum of the time-allocations assigned to the clients being served by that server must be less than the cycle time of the server.* This is captured by the expression:

$$(\forall i)\left(i \notin \texttt{down\_servers}(s) \rightarrow \left(\sum_j \texttt{timealloc}(i,j,s) \le \texttt{cyctime}(i,s)\right)\right).$$

2. *If a server is processing a request for some data, then that data must be available in the server.* This is captured by the expression:

$$(\forall i)(\forall j)\,(m:[b1,b2] \in \texttt{data}(i,j,s) \rightarrow (\forall b)\,(b1 \le b \le b2 \rightarrow i \in \wp(m,b,s))).$$

3. *The sum of consumption rates of the clients being served by a given disk server must not exceed the total disk bandwidth of the server.* This is captured by the expression:

$$(\forall i)(\forall j)\left(\sum_{j:\texttt{timealloc}(i,j,s)>0} \texttt{cons}(j,s) \le \texttt{dtr}(i,s)\right).$$

4. *For each server i that is down, there is no active client.* This is captured by the expression:

$$(\forall i)\,(i \in \texttt{down\_servers}(s) \rightarrow (\texttt{d\_active}(i,s) = \emptyset)).$$

The above constraints specify the basic constraints that tie together, the resources of the VoD disk server system, and the requirements of the clients.

3.2 Prioritized Events

Informally speaking, an *event* is something that (potentially) causes the VoD system to make a transition from its current state to a "next" (or new) state. The study of the performance of disk servers for multimedia applications varies substantially, depending upon the space of events that are considered in the model. In our framework, the space of events that are allowed falls into two categories:

- **Client events:** `enter,exit,pause,play,fast-forward,rewind`;
- **System events:** `server-up, server-down, insert, delete`.

Each event has an associated integer called the *priority* of the event, and a set of attributes. For example, the event `server-up` has an attribute specifying which server is up. Thus, `server-up(2,s)` specifies that the event "server 2 is up" has occurred at state s, while the statement `server-down(3,s)` specifies that the event "server 3 has gone down" has occurred at state s. Likewise, the event `insert` has three attributes – a server id, movie id, and a block number, specifying which block of which movie is being inserted and to which server this insertion is being made. For example, the event `insert(2,m1,b1)` specifies that block $b1$ of movie $m1$ is being inserted onto server 2.

The priority of the event indicates how important the event is – the higher the priority, the greater the importance of the event.

The occurrence of an event must be handled by the VoD system, by *transitioning* to a new state that appropriately "handles" the event. Before specifying how events are handled, we specify some new concepts.

3.3 Modeling Usage Constraints

In any VoD system, the system administrator may wish to enforce some "usage" constraints. In this paper, we do not try to force constraints upon the system. However, we do make available to the system administrator, the ability to articulate and enforce certain types of constraints that s/he feels are desirable for his system.

- **Pause time constraint:** A pause time constraint associates, with each client c, an upper bound, $\texttt{pause\_win}(c, s)$, on the amount of time for which the customer can "pause" the movie s/he is watching. For example, suppose $\texttt{pause\_win}(\text{John Smith}, s) = 25$. This means that as far as the system is concerned, John Smith's pause time cannot exceed 25 time units at state s. In particular, if John Smith pauses the movie he is watching at time 75, then his pause window will expire at time $75 + 25 = 100$, and the resources allocated to him by the VoD system will be "taken back" by the system to satisfy other users' requests.

  In general, when a customer "pauses", the server(s) satisfying the customer's request continues to 'hold" the resources needed by the system. Clearly, holding such resources for an indefinite period is not wise. The pause window specifies, for each customer, an upper bound on the period of time for which the customer can pause the movie.

- **Fast-forward/Rewind window constraint:** Just as in the case of pause windows discussed above, each client c has an associated *fast forward* and *rewind* window which specifies an upper bound on how many data blocks the client can fast-forward to or rewind to, respectively. We use the notation $\texttt{ff\_win}(c, s)$ and $\texttt{rew\_win}(c, s)$ to denote the fast-forward and rewind windows associated with client c at state s.

- **Playing time constraint:** In addition to the pause window, the system manager may wish to specify that a user cannot watch a movie for arbitrarily long. As an example, consider our client John Smith, watching movie $m1$ which has 4 blocks. John Smith's transactions could be highly redundant if, for instance, he were to execute the following transactions:

    1. watch blocks b1,b2,b3
    2. rewind to block b1. Return to step 1

  In order to prevent "irresponsible" usage such as the above, the system manager may specify a *total watch window* for each user and any given movie. For example, specifying that $\texttt{watch\_win}(\mathcal{M}_1, \text{John Smith}) = 180$ says that that John Smith has at most 180 time units to finish viewing movie $\mathcal{M}_1$.

3.4 Update Boundaries

Suppose s is a system state (at some arbitrary point in time) and j is a client being served by a server i. As usual, the state s contains a `data` tuple specifying what data is being provided to the client by that server. For example, consider the situation described in the example of Section 3.1. In that example, in the state shown, server 1 is presenting blocks b2 and b3 of movie $m1$ to client c_1.

Now, suppose the system administrator wishes to delete block $b1$ of movie $m1$ on server 1. Figure 6 shows this situation. *While the system manager has the ability to make the request at any time, the precise time at which the request is actually scheduled (i.e. the precise time at which deletion of the block is scheduled) must take into account, the existing clients watching that movie w.r.t. the server in question.* In this case, the question that needs to be addressed is: *What happens if the client c_1 wishes to rewind 1 block?* If the deletion is incorporated immediately upon receipt of the deletion request, then the rewind request of the client will be denied – a situation that may or may not be desirable. Thus, at any given point in time, each client has an associated *rewind boundary* associated with each server, specifying "how far back" that server can support a rewind request issued by the client. The rewind boundary may change with time. Rewind boundary, and their dual concept of a similar *fast-forward boundary*, are defined below.

Definition 3.1 (Rewind Boundary) The *rewind boundary* of a movie m w.r.t. server i in state s is defined as follows*:*

$$\begin{aligned}&\textit{Rewind\_Boundary(m,s)}\\&\quad = \underline{\textit{min}}\ \{\ b \text{ - } \texttt{rew\_win}(j,s)\ \mid\ j \in \texttt{m\_active}(m,s) \textit{ and } (m,[b{:}b']) \subset \textstyle\bigcup_k \texttt{data}(k,j,s)\ \}\end{aligned}$$

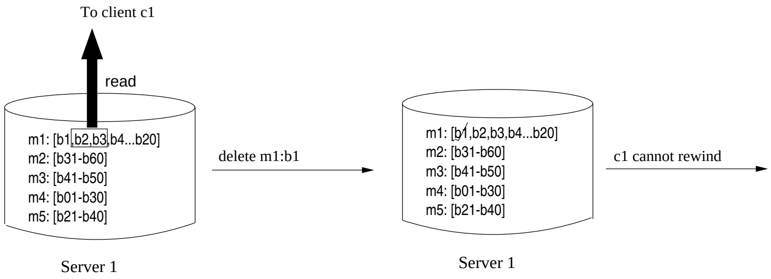


Figure 6: Deletion of a block

If the above set over which the $\underline{min}$ is performed is empty, that is, `m_active`(m,s) is empty, then Rewind_Boundary(i,m,s) = **bnum**(m).

$\diamond$

For example, let us return to the movie $m2$ at server 2 and the state s in which:

1. client c_5 is reading block $b4$ of movie $m2$;
2. client c_6 is reading block $b3$ of movie $m2$;
3. no other client is reading movie $m2$ (exactly what they are doing is not pertinent for this example).

If the rewind window for client $c5$ is 2, and that of client c_6 is 1, then the rewind boundary associated with server 2, movie $m2$ and state s is given by

$$\underline{\min}(4-2, 3-1) = 2.$$

Let us try to see why this is the case, and what this statement means. (Figure 7 illustrates this reasoning).

- Two clients, viz. c_5, c_6, are reading (parts of) movie $m2$ from disk server 2. If we try to update the copy of movie $m2$ residing on disk server 2, the only clients who can be affected (in the current state) are therefore clients c_5 and c_6.
- Client c_5 is currently reading block $b4$ and his rewind window is of length 2, which means he can only go "back" 2 blocks in the movie by executing a rewind command. Effectively, this means that he cannot access any blocks before block $b2$.
- Likewise. client c_6 is currently reading block $b3$ and his rewind window is of length 1, which means he can only go "back" 1 block in the movie by executing a rewind command. Effectively, this means that he cannot access any blocks before block $b2$.
- As the minimum of these two blocks is $b2$, this means that neither client has read access to block $b1$ in this state.

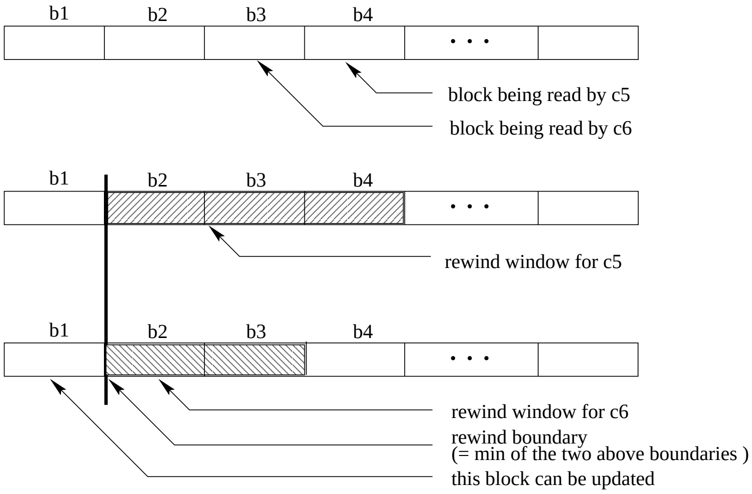


Figure 7: Rewind boundary computation

- Thus, if we wish to update block 1 which lies "below" this rewind boundary, then this is "safe."

An analogous situation occurs w.r.t. *fast forward boundaries* which are defined as stated below.

Definition 3.2 (Fast forward Boundary) The *fast forward boundary* of a movie m in state s is defined as follows*:*

$$\begin{aligned}&\mathit{FF\_Boundary(m,s)}\\&\quad = \underline{\mathit{max}}\ \{\ b + \texttt{ff\_win}(j,s)\ \mid\ j \in \texttt{m\_active}(m,s) \textit{ and } (m,[b{:}b']) \subset \textstyle\bigcup_k \texttt{data}(k,j,s)\ \}\end{aligned}$$

If the above set over which the $\underline{\mathit{max}}$ is performed is empty, that is, $\texttt{m\_active}(m,s)$ is empty, then FF_Boundary*(m,s) = 0.*

◇

For example, consider the single disk server in Figure 8. This disk server, i, contains several movies, but only one of these, viz. movie $m4$ is shown in the figure. Blocks 1–5,7–20 of this movie are available on the disk server i. Suppose that in state s, we have four clients watching this particular movie (other clients may be watching other movies) and that the blocks these clients are watching and the fast forward windows of these clients are as given below:

Client	Block being watched	`ff_win`
c_1	5	2
c_2	9	1
c_3	10	2
c_4	8	3

Then, the fast forward boundary is given by:

$$FF\_Boundary(i, m4, s) = \underline{\max}\{5 + 2, 9 + 1, 10 + 2, 8 + 3\} = 12.$$

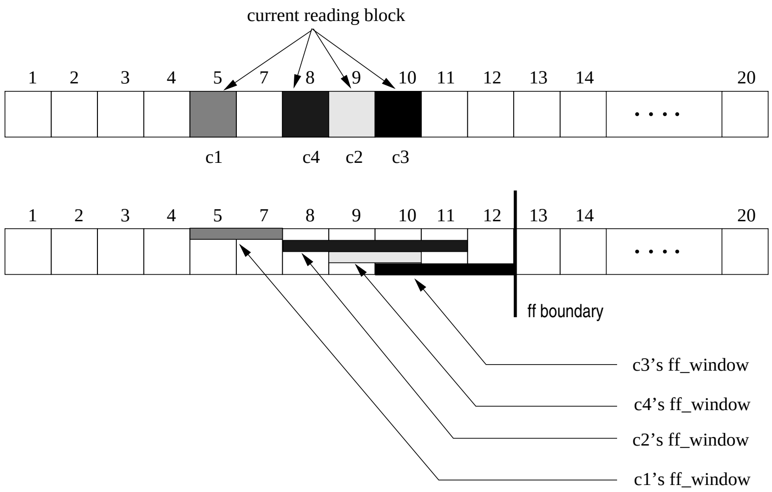


Figure 8: Fast forward boundary computation

This means that only blocks 13–20 of the movie may be updated at this point of time.

The primary use of rewind boundaries and fast forward boundaries is to ensure that when an update request is made by the system manager, that the users viewing the application have the flexibility to rewind or fast forward, within the limits of their fast forward/rewind boundaries. Notice that it is not always possible to guarantee this. For example, in figure 8, if client c_1 wishes to fast forward to block 6, there is no way to satisfy this request without switching him to another disk server, because the disk server in figure 8 does not have block 6.

With these definitions in mind, we are now ready to define how to handle events.

4 Handling Events

In this section, we provide detailed algorithms for handling events. We will first provide an abstract, declarative specification of *what* constitutes an appropriate way of handling events. Then, we will provide algorithms to successfully handle events.

4.1 Optimal Event Handling: Specification and Semantics

Suppose s is a valid state of the system, and e is an event that occurs. In this section, we will first specify what it means for a state s' to handle the event e occurring in state s. This will be done without specifying how to find such a state s'. We will later provide algorithms to handle such events.

Definition 4.1 (Event Handling) State s' is said to *handle* event e in state s iff one of the following conditions is true:

1. **New clients:** [$e =$New client c enters with a request for movie m:]
   $(\exists i)(i \in \mu_{s'}(c) \wedge m : [1,1] \in \texttt{data}(i,c,s')$.

2. **Old clients:** [$e =$Old client c exits the system]
   $\mu_{s'}(c) = \emptyset$.

3. **Continuing clients:**

   (a) ($e =$Continuing client c watches, in "normal viewing" mode, block b of movie m)
   $(\exists i)(i \in \mu_{s'}(c) \wedge m : [b,b] \in \texttt{data}(i,c,s'))$.

   (b) ($e =$Continuing client c pauses)
   $(\exists i)(i \in \mu_{s'}(c) \wedge \texttt{data}(i,c,s') = \emptyset$.

   (c) ($e =$Continuing client c fast forwards from block b to block $b+r$ where $r \leq \texttt{ff\_win}(c,s)$)
   $(\exists i)(i \in \mu_{s'}(c) \wedge m : [b, min(\textsf{bnum}(m), b+r)] \in \texttt{data}(i,c,s')$.

   (d) ($e =$Continuing client c rewinds from block b to block $b-r$ where $r \leq \texttt{rew\_win}(c,s)$)
   $(\exists i)(i \in \mu_{s'}(c) \wedge m : [max(0, b-r), b] \in \texttt{data}(i,c,s')$.

4. **Server status event:**

   (a) ($e =$disk server i crashes)
   $i \in \texttt{down\_servers}(s') \wedge \neg\texttt{server\_status}(i) \wedge (\forall c) i \notin \mu_s(c)$.

   (b) ($e =$disk server i comes back "up")
   $\texttt{up\_server}(i) \wedge \neg\texttt{down\_servers}(i)$.

5. **Update event status:**

   (a) ($e =$ delete block b of movie m from server i)
   $i \notin \wp(m,b,s') \vee (i \in \wp(m,b,s') \wedge (i,m,b) \in \texttt{delete\_list}(s')))$.

   (b) ($e =$ insert block b of movie m into server i)
   $i \in \wp(m,b,s') \vee (i \notin \wp(m,b,s') \wedge (i,m,b) \in \texttt{insert\_list}(s'))$.

The handling of *update events* requires some intuition. Let us suppose, that we have a movie containing 100 blocks which is stored, in its entirety, on one disk server, and we have 2 clients c_1, c_2 who are watching the movie, via this server. Let us say that c_1 is watching block 45, and c_2 is watching block 50, and each of them is consuming 1 block per time unit (just to keep things simple). Let us further say that the system manager now wishes to update the entire movie, replacing old blocks by new ones (which may be viewed as a simultaneous insert and delete). Additionally, both clients c_1, c_2 have rewind windows and fast forward windows of 5 blocks each. Figure 9 shows this situation.

- At this stage, the rewind and fast forward boundaries for this movie are 40 and 55, respectively.
- This means that blocks $1, \ldots, 39$ and $56, \ldots, 100$ may be safely updated right away (assuming that enough bandwidth is available).
- The blocks b such that $40 \leq b \leq 55$ can only be updated later, i.e. the updating of these blocks must be *deferred*.

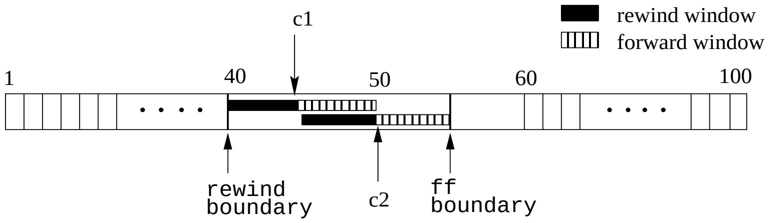


Figure 9: Example of deferred updates

- For example, after one time unit, block 40 can be updated. After 2 time units, block 41 can be updated, and so on.

The skeptical reader will immediately wonder whether this definition allows us to postpone update events for an arbitrarily long time. The answer is that *as stated above, update events could get deferred for ever.* To avoid this situation, and to also assign different priorities to different clients, we now introduce the notion of *priority.* Associated with each event (client initiated or system initiated, or deferred) is a priority. The higher the priority, the more important the event. In particular, if e is an update event, and e is deferred, then for each time unit that e is updated, we must "increment" e's priority by a factor δ_e. Thus, different events can have different associated "prioritization steps" which may be selected by the system manager, based on the importance of the event as determined by him/her. What this means is that the priority of an update events "gradually rises" till it can be deferred no longer. We discuss this scheme in detail below, and also show how the same idea applies to priorities on other (non-deferred) events.

4.2 Priority Scheme for events

Whenever an event occurs, that event is assigned an initial priority, either by the system, or by the system administrator. The system maintains a list of default priority assignments. In the event of a different priority assignment being made by the system administrator, then the latter overrides the former.

Integers are used to represent "initial" priority assignments, though as we shall see, "non-initial" priority assignments may be real-valued. The precise integers used for initial priority assignment are not really important. What is more important is the *relative* priority ordering.

Default Initial Priority Assignments: Figure 10 shows the initial priority assignments. The rationale for these assignments is discussed below.

1. System events have the highest priority. The reason for this is that a server crash, or a server coming "back up" are events that are hard to control. It is not possible, for instance, to defer or delay a crash. If it occurs, the system must transition to a new state that "handles" the crash as best as possible.
2. Next, existing clients already being served by the system must have the highest priority. The reason for this is that the VoD system has made a *commitment* to serve these clients well, and

Event Type	Event	Priority
System	Server_down	9
System	Server_up	9
Client (old)	Exit	7
Client (continuing)	Pause	6
Client (continuing)	Play	5
Client (continuing)	Fast - Forward	4
Client (continuing)	Rewind	4
Client (enter)	Enter	3
Manager	Delete	2–7
Manager	Insert	1

Figure 10: Initial assigned priorities for different events

it must try to honor these commitments. However, each existing client may "spawn" different events, including `exit`, `pause`, `play`, `fast forward`, and `rewind`. Each of these events has a different priority.

(a) The highest priority is assigned to events that `exit`. Processing an `exit` event early is desirable in general, because this can be done very fast, and furthermore, this frees up resources that may be used to satisfy other clients (continuing clients, as well as potential new clients).

(b) The next highest priority in this class is assigned to `pause` events because: first these events request no new resources (and hence, they can be satisfied immediately) and second, because of the `pause` window, these events may lead to future `exit` events that do in fact free up resources.

(c) The next highest priority in this class is assigned to `play` events. The reason for this is that in most cases, play events are relatively easy to satisfy as they merely require that the next block of the movie be fetched, and in most cases, the next block will be on the disk(s) that are already serving the client.

(d) The last two events in this category, with equal priority, are `rewind` and `fast forward`. These events may require substantial "switching" of clients (i.e. a client may be switched from its current server to another, because when blocks are skipped, the current server no longer has blocks that are several "jumps" ahead of the block currently being scanned.

3. Of the system events, the `delete` event has the highest priority. The reason for this is that delete events can be accomplished by a very simple operation – just remove the pointers to the appropriate blocks. In contrast, `insert` events require greater resources (e.g. disk bandwidth is need to write onto the disk).

Priority Steps: Suppose an update request is received for block b of movie m in server i. Furthermore, suppose rwb and fwb denote, respectively, the rewind boundary, and the fast forward boundary associated with the current state. It is not difficult to see that we must have $rwb \leq ffb$. The update cannot be carried out immediately if $rwb \leq b \leq ffb$. As a consequence, we might need to *defer* the update. However, as mentioned above, deferring the update might cause the update to be indefinitely delayed. To ensure that this does not happen, the system administrator may associate with each update request u, a *priority-step*, δ_u. δ_u is a non-negative real number, and its interpretation is as follows:

- Suppose s_0 is the current state (in which the update u occurs with the priority p shown in Figure 10 (p must be either 1 or 2).
- Suppose $s_1, s_2, \ldots, s_k$ are states that occur, consecutively after s_0, all of which defer update u.
- Then the priority p_i of the update event u in state s_i is $(p + i \times \delta_u)$.

Thus, for example, suppose u is a deletion request, and the system manager assigns a step of 0.2 to u. Then, after 6 state changes (i.e. in state s_6), the priority of this update will be 3.2, which would exceed the priority of a new event (which is 3) occurring in that state. What this means is that if a new client enters the system in state s_6, and requests a movie, then the server in question would be asked to consider the higher priority update request u, as opposed to serving the customer.

By making the step size small, the system manager can allow a greater period of time to elapse before making the update have higher priority over new clients. For example, had the system manager set δ_u in the above example to 0.002, then 501 state changes would have to occur, before update u's priority exceeded that of a new client.

Furthermore, the system manager does not have to specify the same priority step for each update. Different updates can have different associated priorities, as would be expected in most real life systems.

We are now ready to give an algorithm that manipulates the priorities, such as those shown in Figure 10, and the above priority steps, to handle events that occur at any given point in time.

Video Server with Updates and Crashes (VSUC) Algorithm

```
main HandleEvents ( NewEvents, OldEvents )
{
    EvtList = sort events in NewEvents and OldEvents in decreasing order of priority ;
    WaitList = ∅ ; /* set of events that can't be scheduled in this cycle */
    DoneList = ∅ ; /* set of client events that have been scheduled successfully */

    While ( !timeout and EvtList ≠ ∅ )
    {
        evt = get the first event in EvtList ;
        switch ( evt.type )
        {
            case down :   handleServerDownEvents ( evt )
```

```
            case up :  handleServerUpEvents ( evt )
            case play, rewind,  fast-forward :  handleContEvents ( evt )
            case pause :  handlePauseEvents ( evt )
            case exit :  handleExitEvents ( evt )
            case enter :  handleEnterEvents ( evt )
            case insert:  handleInsertEvents ( evt )
            case delete:  handleDeleteEvents ( evt )
        }
    }

    If ( EvtList ≠ ∅ )
        increase priority of each event in EvtList by δ_evt ;

    OldEvents = merge events from EvtList and WaitList ;
    return ( OldEvents ) ;
}

procedure HandleServerDownEvents ( evt )
{
    for each data block b_i in crashed server do
        update placement mapping so that b_i is not visible ;

    for each event e_i in crashed server do
        insert e_i into EvtList preserving the sorted order ;
}

procedure HandleServerUpEvents ( evt )
{
    for each data block b_i in recovered server do
        update placement mapping so that b_i is visible ;
}

procedure HandleExitEvents ( evt )
{
    release resources and data structures allocated for evt ;
}

procedure HandleContEvents ( evt )
{
    Blocks = set of blocks necessary for servicing evt ;
    /* depending on event type, the way blocks are read from disks can be */
    /* different.  For example, in play event, certain number of continuous */
    /* blocks should be read, but in rewind(ff) event, some intermediate blocks */
    /* can be skipped to match the speed */

    if ( servers that have been assigned to evt contain all blocks in Blocks )
    {
        update the data component of evt ;
        insert evt into DoneList ;
        return ;
    }

    DServers = set of servers that contain all blocks in Blocks ;
```

if ($DServers = \emptyset$) /* placement mapping error */
{
    /* make evt considered after block insertions */
    decrease evt's priority by δ_{evt} ;
    insert evt into $EvtList$ preserving the sorted order ;
    return ;
}

$RServers$ = set of servers in $DServers$ satisfying resource constraints ;
if ($RServers = \emptyset$)
{
    if (evt's priority has been decreased previously)
    {
        $Finished$ = **false** ;
        $Svlist = DServers$;
        while ($Svlist \neq \emptyset$ and $!Finished$) do
        {
            s = select one server randomly from $Svlist$;
            $Svlist = Svlist$ - { s } ;
            $Switchables$={e | event e is served by s and there exists $s'(\neq s)$ that satisfies e } ;
            while ($Switchables \neq \emptyset$ and $!Finished$) do
            {
                e' = select one event randomly from $Switchables$;
                $Switchables = Switchables$ - $\{e'\}$;
                if (evt can be served using the resources that will be released from e')
                {
                    release resources from e' and update resource allocation of s ;
                    allocate resources to evt and update resource allocation of s ;
                    allocate resources to e' and update resource allocation of s' ;
                    put evt into $DoneList$;
                    $Finished$ = **true** ;
                }
            }
        }

        if ($!Finished$)
        {
            /* make evt scheduled prior to other clients in next cycle; */
            increase evt's priority by δ'_{evt} ;
            insert evt into $WaitList$;
        }
    } else
    {
        /* make evt considered after scheduling other normal continuing clients
*/
        decrease evt's priority by δ''_{evt} ;
        insert evt into $EvtList$ preserving the sorted order ;
    }
} else
{
    $MaxEval$ = - 1 ;

```
        for each server s_i in RServers do
        {
            Eval = evaluate s_i for the specified criteria ;
            if ( Eval > MaxEval )
            {
                MaxEval = Eval ;
                BestSv = s_i ;
            }
        }
        allocate resources to evt from BestSv ;
        update resource allocation of BestSv ;
        insert evt into DoneList ;
    }
}

procedure HandlePauseEvents ( evt )
{
    yield disk bandwidth to update events for next cycle ;
    keep the other status unchanged ;
}

procedure HandleEnterEvents ( evt )
{
    /* enter event can be handled in a way similar to handling continuous events.  */
    /* The difference is that in the case of enter events, resources have not */
    /* been assigned previously.  Therefore, checking if already assigned server */
    /* can handle the event is not necessary for enter events.  */
}

procedure HandleInsertEvents ( evt )
{
    s_evt = server that data block is inserted into ; /* specified in evt */
    Dsize = the size of data that is inserted into s_evt ;
    Msize = maximum data size that server s_evt can handle using available resources ;
    if ( Msize ≥ Dsize )
    {
        allocate resources to evt ;
        update resource allocation of s_evt by Dsize ;
        update placement mapping information of s_evt ;
    } else
    {
        /* Dsize can't be inserted in its entirety */
        allocate resources to evt ;
        update resource allocation of s_evt by Msize ;
        reduce evt's data size by Msize ;
        increase evt's priority by δ_evt ;
        insert it into WaitList ;
    }
}

procedure HandleDeleteEvents ( evt )
{
```

```
    b_evt = block number that is deleted ;
    calculate the rewind and fast forward boundary of the movie ;
    if ( b_evt < rewind_boundary or b_evt > ff_boundary )
        delete b_evt and update placement mapping information ;
    else
    {
        /* evt is deferred to next cycle */
        increase evt's priority by δ_evt ;
        insert it into WaitList ;
    }
}
```

It is easy to prove that the VSUC algorithm described above has a number of nice properties, as stated in the theorems below. An informal description of these properties is as follows:

- *Under certain reasonable conditions, clients who have already been admitted to the system experience no jitter, independently of what other events occur.* This result applies when (1) if the placement mapping is "full" (i.e. either the entire movie is available through a server, or none of it is), and (2) when the client watches a movie entirely in "normal" viewing mode, and (3) no server outages occur.
- *Every event eventually gets handled as long as servers that go "down" eventually come back "up."*
- *The VSUC algorithm runs in polynomial time, i.e. if the current state is s and if ev is the set of events that occur, then a new state s' (together perhaps with deferred events) is computed in polynomial time.*

Theorem 4.1 (Continuity of Commitments) *Suppose s is the current state of the system, and C_i is a continuing client in state s who is watching movie m in "normal" mode. Furthermore, suppose that:*

1. *movie m is contained in its entirety in each server $sv \in \mu_s(i)$ and*
2. *no server in $\mu_s(i)$ goes "down" at this time and*
3. *for all updates u (before client C_i entered the system,) that were deferred when client C_i enters, $pr_u \leq 5$ and $\delta_u \leq \delta_{C_i}$ where pr_u is the priority of the update u when client C_i enters the system, δ_u is the priority step associated with the update, and δ_{C_i} is the priority step associated with C_i.*
4. *for all updates u (before client C_i entered the system,) that enter the system after client C_i enters, $newpr_u \leq 5$ and $\delta_u \leq \delta_{C_i}$ where $newpr_u$ is the priority of the update u when it enteres the system.*

Then client C_i's movie request event will be satisfied by the VSUC algorithm.

Proof Sketch. In the VSUC algorithm, the only event that diminishes the system's resources and that has a higher priority than a continuing client is a `Server_down` event or a deferred update event. However, by the assumption in the statement of the theorem, no servers serving client C_i go down, and hence, the highest priority events are either deferred updates or continuing clients.

Suppose a server sv is serving client C_i's request (in part or in full). If no deferred events occur, then the same server can continue servicing client C_i's request for "next" blocks. However, if deferred events occur, then there are two possibilities:

1. Suppose the deferred update u was requested before client C_i entered the system. As $pr_u \leq 5$ and as $\delta_u \leq \delta_{C_i}$, it follows that throughout the normal playing of the movie, client C_i's priority is higher than that of the update u. Thus, server sv continues to serve client C_i without allowing deferred events to obtain priority over the client C_i.
2. On the other hand, if the deferred update was requested after client C_i entered the system, then client C is guaranteed to obtain priority over the update because $newpr_u \leq 5$ and as $\delta_u \leq \delta_{C_i}$. Hence, client C_i can continue to be served by server sv. □

The above theorem has important implications for **admission control**, both of new clients and of new updates.

- **Client Admission:** To guarantee continuity of service, a new client C_i should be admitted to the system *only if* for all deferred updates u that need to be handled when client C_i enters the system, we must know that $pr_u \leq 5$ $\delta_u \leq \delta_{C_i}$.
- **Update Admission:** To guarantee continuity of service to existing clients, a new update u should be admitted to the system only if $newpr_u \leq 5$ and as $\delta_u \leq \delta_{C_i}$.

Theorem 4.2 (All update events get handled eventually) *Suppose s is the current state of the system and ev is any update event that requires a set SV of servers. Further suppose that for all times $t > now$ and all servers in SV, if there exists a time $t' > t$ at which one or more servers in SV go down, then there exists a time $t^\star > t'$ at which all servers in SV come back up. Then: for any update event ev that occurs now, there exists a time $t_{ev} \geq now$ such that ev gets handled at time t_{ev}.*

Proof Sketch. If update event ev does not get handled *now*, then, as $\delta_{ev} > 0$, in each execution of the VSUC algorithm, event ev's priority *strictly increases* till it exceeds 7, at which point t' in time, it will be handled unless one or more servers that are needed to service event ev are down. By the restriction in the statement of the theorem, there exists a time $t^\star > t'$ at which all servers in SV are "up" simultaneously. We are guaranteed that this event will be handled latest at time $t^\star$. □

Theorem 4.3 *Suppose $ev(t)$ is a set of events that occur at time t. The time taken for the the VSUC algorithm to terminate is polynomial in the sum of the number of events in $ev(t)$ and the number of deferred events.*

Proof. It follows immediately that each function call in the main algorithm runs in time polynomial w.r.t. the above sum. □

1	Total Number of Video Clips number of 10 minutes video number of 20 minutes video number of 40 minutes video number of 80 minutes video	800 400 200 100 100
2	Size of Video Segment	10-80 minutes
3	Size of Block	0.2 seconds' compressed video data
4	Number of Requests	800-2000
5	Request Pattern	Based on actual data referenced in [5]
6	Number of Disk Servers	30
7	Types of Disk Servers	4
8	Buffer size	Avg. 50 MB per server
9	Disk Bandwidth	Avg. 20 MB combined per server

Table 1: parameters Used For Simulation

5 Experiments

5.1 Crash Handling vs Survival rate

In this paper, our video server consists of multiple disk servers with possibly different performance characteristics. Table 1 shows several parameters related to the experiment.

In the first set of experiments, we examined the resilience of our video server against disk server crashes, i.e. how well does our video server perform when crashes occur? To compare, we used three different types of disk server configurations – homogeneous disks of evenly high performance characteristics, homogeneous disks of evenly low performance characteristics, and heterogeneous disk servers [3].

We generated client requests and disk crashes randomly (but in accordance with certain parameters described below) for each case. To compare the resilience of our video server against server crashes, we first measured the average number of continuing clients. Disk server crashes were generated randomly in respect to crashed server or crash time. Also, the crashed server will eventually be fixed or replaced and put into operation. We used same crash recovery time during experiment. To see the effect of server crashes on the system, we changed the frequency of server crashes measuring continuing clients.

Figure 11 shows the effect of server crash handling on the number of continuing clients. Regardless of disk server configuration, our crash handling approach supported more streams than without crash handling. However, depending upon the performance characteristics of the servers involved, difference numbers of continuing clients could be supported; the most notable improvement was measured in the case of the homogeneous, high-performance disk configuration.

As the frequency of disk crashes increases, the system will experience much more difficulty

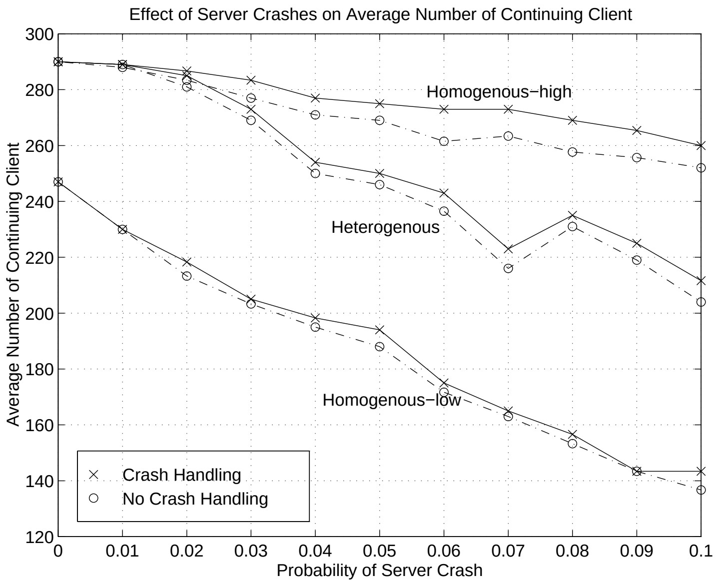


Figure 11: Effect of the server crashes on average number of continuing clients.

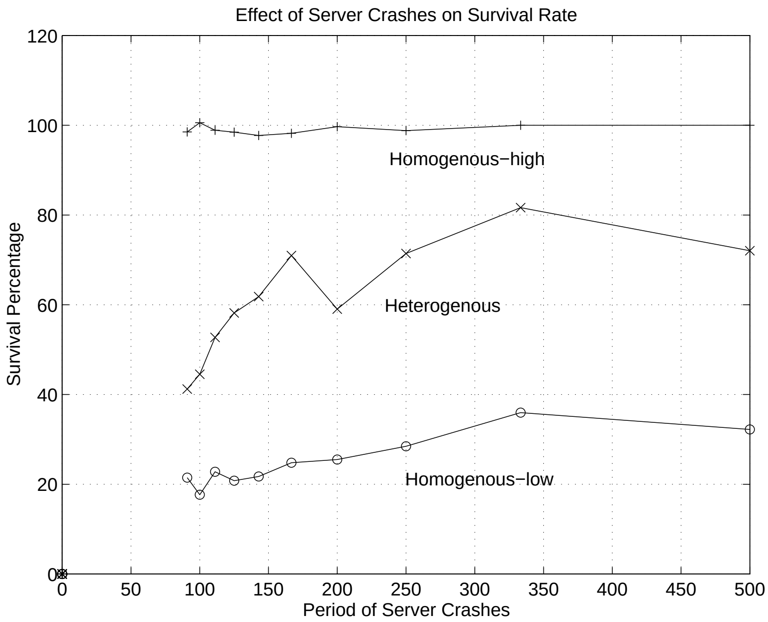
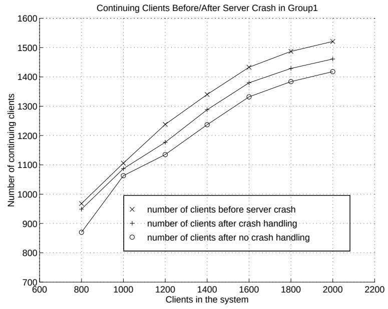


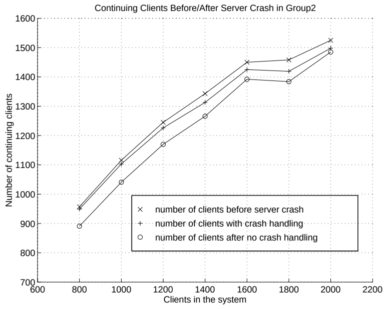
Figure 12: Effect of the server crashes on survival rate.

scheduling clients because disk or buffer resources and video data at the crashed servers are not available during crash recovery time. We measure the survival rates of clients when crashes occurred. Here, survival rate refers to the ratio of clients who continue to be served, as compared to the total number of clients in the system. As is easily seen from figure 12, both systems with homogeneous capacity disk servers showed a stable survival rate w.r.t. disk crashes. However, in the case of heterogeneous capacity disk servers, there was a noticeable fluctuation in the survival rate.

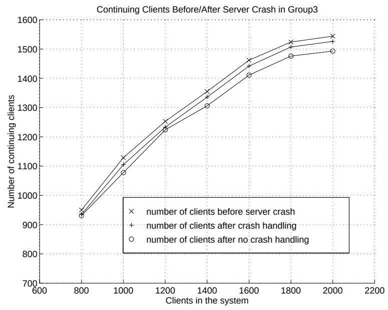
In the second experiment, we used heterogeneous disk servers and examined the effect of crashes on different capacity disk servers. For the experiment, we assumed four different groups of disk servers with different server capacity. Servers with the highest capacity belong to group 1 and servers with the lowest capacity belongs to group 4. Under normal operation, servers with higher capacity store more video segments and provide more concurrent streams to clients than those with lower capacity. Therefore, the effects of disk crashes will vary depending on the capacity of the server that crashes.



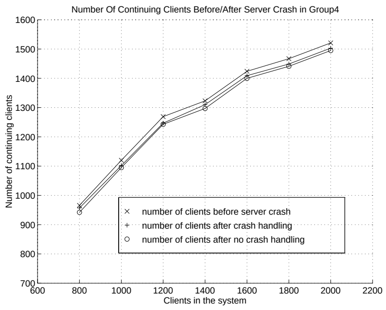
(a) number of continuing clients in Group 1



(b) number of continuing clients in Group 2



(c) number of continuing clients in Group 3



(d) number of continuing clients in Group 4

Figure 13: continuing clients after server crash

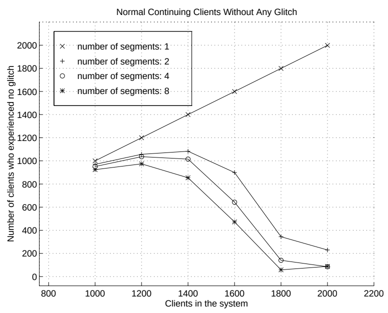
Figure 13.a to 13.d show how many clients on the crashed server continue to be served even after disk crash (under varying system load). To facilitate comparison, we showed together both the number of continuing clients after crash handling and without crash handling. Here, "without crash handling" means that the streams on the crashed server(s) will be discontinued unconditionally.

Figure 13.a shows the effect of crash handling when the crashed disk server belongs to group 1. In this figure, the difference between the top line and the bottom line is the number of clients on the

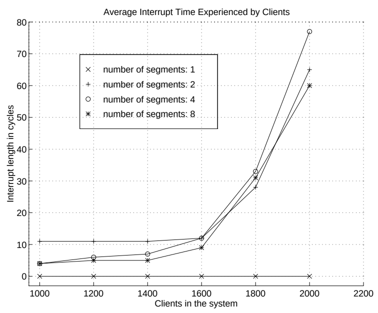
crashed disk. On the average, our crash handling VoD server algorithm can satisfy about half the clients affect by the crash by rescheduling their streams to other available servers.

5.2 Performance vs Segmentation

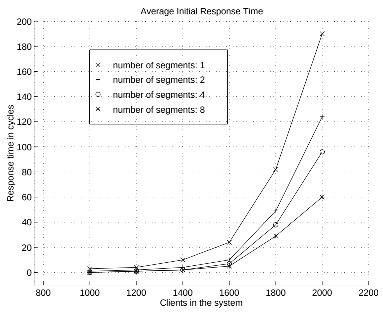
In this experiment, we examined the performance of the video server for different segmentations – here a segment refers to a continuous sequence of video blocks. We assumed that video objects are divided into several segments of equal size. These segments are placed in the disk servers in a way that adjacent segments should be placed in the different disk servers (otherwise multiple segments are merged into one large segment on a single server). Video segments were placed on the servers in a manner proportional to the size of the disk storage available,i.e. the probability that a video segment is placed on a disk having capacity 5 GB is 5 times the probability that the vide segment in placed on a 1 GB disk. Under this segment placement scheme, any two disk servers with adjacent segments should be synchronized for the continuous display of video. That is, as soon as a segment is consumed from the first server, the next segment should be delivered from the second server without delay. If the second server can't deliver next segment stream, then clients may experience degradation in quality. This constraint can be relaxed if we increase buffer space for each stream.



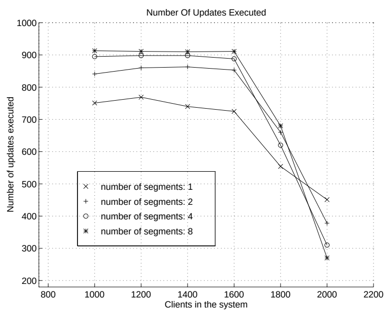
(a) number of continuing clients



(b) average blocked time



(c) average initial waiting time



(d) number of updates executed

Figure 14: experiment results

Figure 14.a shows how many streams will experience intermediate stream delay due to server

switches for retrieval of adjacent segments. When video objects are stored in their entirety, then there is no need for server switches for the ongoing streams. But as the number of segments are increased, the number of clients experiencing intermediate stream delays due to server switch is increasing.

In Figure 14.b, we examined average intermediate stream delay experienced by the clients. It shows that once video objects are segmented, the average stream delay decreases as the number of segments increases. This is mainly due to the fact that smaller segments stay at the server for a shorter time than larger segments. With shorter stays at the servers, resource availability for disk servers increases and therefore server switching can be done more easily and frequently.

¿From figure 14.a and 14.b, we might conclude that storing video objects in its entirety on one disk server is the best scheme. But two other criteria show that this scheme has some disadvantages as well. Figure 14.c shows the server response time specifying how long each client has waited till the first frame of the video object was displayed. Under a moderate to large number of clients in the system, the system response time increases sharply as the number of segments increases. Also, the number of updates done during the simulation time increases as the number of segments increases. Figure 14.d shows how many update requests has been done during the simulation.

Furthermore, as mentioned earlier, intermediate stream delays due to server switches can be compensated to a certain degree if we increase buffer space for streams. From figure 14.b, average stream delay is less than 30 cycles when the total number of clients is 1800. Therefore, as we increase the buffer space for streams, the number of clients experiencing actual intermediate display delay will be reduced.

6 Conclusions

Though there has been extensive work on handling disk crashes most such work has occurred in the area of recovery of data on the crashed disk. Likewise, though there has been extensive work on developing systems support for handling VCR-like functions in video servers, this work has ignored two possibilities:

1. That during the operation of such a video server, *updates* might occur. The problem of handling such updates has not been adequately addressed in the literature.
2. Similarly, during the operation of such a video server, one or more servers might crash and/or otherwise become inaccessible. This means that any clients currently being served by those servers must be satisfied in some other way. To date, there has been no formal theoretical work on extending VoD servers to handle this possibility.

The primary aim of this paper is to provide a formal model of VoD systems that is capable of handling such events, as well as to provide the VSUC algorithm that can neatly handle the variations in resource availability that may arise as a consequence of such events. In particular, the VSUC algorithm has many nice properties that, to our knowledge, have been proposed for the first time.

- First, the VSUC algorithm guarantees that under certain reasonable conditions, users to whom the VoD server has already made commitments, will experience no disruption or jitter in service as long as they watch the movie in "normal" mode.
- Second, the VSUC algorithm guarantees (again under certain reasonable restrictions) that no request made by a continuing client will be denied service "forever", i.e. it will eventually be handled.
- Third, the VSUC algorithm reacts to events, both user-initiated, and system-initiated, in polynomial time.

Acknowledgements

This work was supported by the Army Research Office under Grants DAAH-04-95-10174 and DAAH-04-96-10297, by ARPA/Rome Labs contract F30602-93-C-0241 (ARPA Order Nr. A716), by Army Research Laboratory under Cooperative Agreement DAAL01-96-2-0002 Federated Laboratory ATIRP Consortium and by an NSF Young Investigator award IRI-93-57756. We are grateful to Dr. B. Prabhakaran for a careful reading of the manuscript and for making many useful comments and critiques.

References

[1] S. Berson and S. Ghandeharizadeh. (1994) *Staggered Striping in Multimedia Information Systems*, Proc. 1994 ACM SIGMOD Conf. on Management of Data, Minneapolis, MN, pps 79–90.

[2] S. Berson, L. Golubchik and R. Muntz. (1995) *Fault Tolerant Design of Multimedia Servers*, Proc. 1995 ACM SIGMOD Conf. on Management of Data, San Jose, CA, pps 364–375.

[3] K.S. Candan, E. Hwang and V.S. Subrahmanian. *An Event-Based Model for Continuous Media Data on Heterogeneous Disk Servers*, ACM Multimedia Systems Journal, accepted, to appear.

[4] M.-S.Chen, D.D. Kandlur, and P.S. Yu. (1994) *Support for Fully Interactive Playout in a Disk-Array-Based Video Server*, Proc. ACM Multimedia 1994, pps 391–398.

[5] A. Dan and D. Sitaram, "A Generalized Interval Caching Policy for Mixed Interactive and Long Video Workloads", *Multimedia Computing and Networking*, San Jose, January 1996.

[6] A.L. Drapeau, D.A. Patterson, and R.H. Katz. (1994) *Toward Workload Characterization of Video Server and Digital Library Application*, ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Nashville, May 1994.

[7] C. Federighi and L. Rowe. (1994) *A Distributed Hierarchical Storage manager for a Video-on-Demand System*, Proc. of the 2nd SPIE Symp. on Storage and Retrieval of Video Databases, pps 185–197.

[8] Shahram Ghandeharizadeh and Cyrus Shahabi. (1994) *On Multimedia Repositories, Personal Computers, and Hierarchical Storage System*, Proceedings of ACM Multimedia 1994.

[9] G. Miller, G. Baber, and M. Gillilana. (1993) *News-on-Demand for Multimedia Networks*, Proc. ACM Multimedia 1993, pps 383–392.

[10] Antoine N. Mourad 1996. *Issues in the Design of a Storage Server for Video-On-Demand*, ACM/Springer-Verlag Multimedia Systems, 1996

[11] A.L. Narasimha Reddy. (1995) *Scheduling and Data Distribution in a Multiprocessor Video Server* , Proceedings of IEEE Multimedia 1995.

[12] D. Patterson, G. Gibson, and R. Katz. (1988) *A Case for Redundant Arrays of Inexpensive Disks*, Proc. ACM SIGMOD Conf. on Management of Data 1988.

[13] C. Ruemmler and J. Wilkes. (1994) *An Introduction to Disk Drive Modeling*, IEEE Computer, pps 17–28, March 1994.

[14] K. Salem and H. Garcia-Molina. (1986) *Disk Striping*, Proc. 1986 IEEE Conf. on Data Engineering.

[15] J.L. Sharnowski, G.C. Gannod, and B.H.C. Cheng. (1995) *A Distributed, Multimedia Environmental Information System*, Proceedings of IEEE Multimedia 1995.

[16] R. Tewari, R. Mukherjee, D.M. Dias, and H.M. Vin, *Design and Performance Tradeoffs in Clustered Video Servers*.

[17] P. Venkat Rangan, H. Vin and S. Ramanathan. (1992) *Designing and On-Demand Multimedia Service*, IEEE Communications Magazine, pps 56–64, July 1992.

[18] H. Vin, S.S. Rao and P. Goyal. (1995) *Optimizing the Placement of Multimedia Objects on Disk Arrays*, Proc. 1995 IEEE Intl. Conf. on Multimedia Computing Systems, pps 158–165.

[19] B. Worthington, G. Granger and Y. Patt. (1994) *Scheduling Algorithms for Modern Disk Drives*, Proc. 1994 ACM SIGMETRICS Conference.