A Failure and Overload Tolerance Mechanism for Continuous Media Servers[1]

R. Krishnan, D. Venkatesh, and T.D.C. Little

Department of Electrical and Computer Engineering
Boston University, Boston, Massachusetts 02215, USA
(617) 353-9877
tdcl@bu.edu

MCL Technical Report No. 03-12-1997

Abstract–Large scale clustered continuous media (CM) servers deployed in applications like video-on-demand have high availability requirements. In the event of server failure, streams from the failed servers must be reassigned to healthy servers with minimum service disruption. Such servers may also suffer from periods of transient overload resulting from a high degree of customer interactivity. For example, in a video-on-demand system if a large number of users are viewing a favorite game, many of them could simultaneously request a replay of an interesting part of the game. This requires a large number of "interactive" channels within a short period of time and can result in a transient server overload. In this paper we propose solutions for graceful recovery from overload scenarios arising out of server failure or customer interactions. Rapid resource reclamation is key to overload tolerance, and our proposed solution is based on rate adaptive stream merging and content insertion techniques. We also utilize conventional time-sharing techniques to handle transient overload. We show that while merging is necessary for achieving overload tolerance, it is not sufficient, and for a complete solution, content insertion is required. Specifically, we consider a general clustered CM server architecture model where multiple servers can fail simultaneously. We develop a model for resource shortfalls that occur as a result of overload on failure. We also describe optimal polynomial time algorithms for recovering resources to the maximum extent possible, by clustering streams in real time.

Keywords: Overload tolerance, fault tolerance, clustered video servers, interactive video-on-demand, content insertion, rate adaptive stream merging, stream clustering, caching.

[1]*Proc. 5th Intl. ACM Multimedia Conf. 1997*, Seattle, WA, USA, Nov 1997. This work is supported in part by the National Science Foundation under Grant No. NCR-9523958, EMC Corporation, and Hewlett-Packard.

1 Introduction

In conventional file servers and operating systems, on failure or on overload, clients wait until service is resumed. This is unacceptable for continuous media (CM) applications. In particular, with video it is necessary that content is continuously available to the viewer even on the event of server failure.

Consider the following analogy of a movie theater. In the event of a projector failure, it would be far more desirable to watch a preview of another movie than to stare at a blank screen, especially if the failure persists for a long time. In other words, it may be acceptable under failure conditions to alter the presentation in some reasonable way that does not significantly affect the users. This indicates that the definitions of fault resilience must be rethought for emerging technologies such as digital video broadcasting, that is, the structure of data can be reorganized and altered to meet a desired quality of presentation, measured by the continuity of presentation. Clearly, this assumption must be validated against the application being evaluated. Inserting arbitrary content would be extremely catastrophic for an application such as medical imaging where the correctness of data is more important than the visual experience. However, a majority of home edutainment (education + entertainment) applications can take advantage of content modification techniques and it is these that we consider in this paper.

Video, as a visual medium, is relatively resilient to data corruption due to the high amount of redundancy in visual information. Fault recovery techniques that take advantage of this observation can be classified into two broad categories: (i) Techniques that alter content and (ii) Techniques that drop information. Fundamentally, both techniques achieve fault resilience by reducing the resources required by a video stream (measured by its bandwidth). Content alteration techniques take advantage of the intra-image redundancy whereas frame-dropping techniques take advantage of inter-image redundancy [8, 9, 10]. An alternate approach is to provide fault resilience by building redundancy into the system as in RAID based servers [2, 9, 8].

Rate-adaptation and stream manipulation require some content to be dropped [6, 10, 3]. Furthermore, these schemes tend to be either CPU intensive or require a significant time to achieve the desired reduction in resource utilization. RAID based techniques to handle overload typically involve over-allocation of resources. In large systems, introducing redundancies for correcting errors as well as for fault tolerance is expensive. Furthermore, over-allocation of resources to handle overload is clearly not economical. It is therefore desirable to develop a scheme that makes efficient use of resources yet recovers gracefully under failure with a minimum impact to the user.

In this paper, we introduce *content insertion* as a means to reclaim resources and recover from overloads arising out of a fault condition. Most of the current work in the field address the overload problem by degrading the quality of service to the user, for example by dropping information content to recover system bandwidth resources. This does not meet the requirements of the media broadcasting industry. Additionally, they do not consider the suitable "reflex-response" that is required to ensure continuity of presentation to the user. We perceive that the paradigm of "sharing" of resources is better than "rationing" of resources. Content insertion is an efficient and practical alternative that provides a mechanism to easily implement sharing of resources. More important, it can be easily adapted to work in conjunction with other the conventional techniques described earlier.

In content insertion, users are presented with an alternative media stream during overload or fault conditions while the fault recovery mechanism is activated. The advantage here is that multiple users can be placed on the same channel thus conserving resources. The new content can contain advertisements or previews and subsidize customer subscription costs.

In this paper, we focus on building such a fault recovery scheme in a storage server, but the solutions can be scaled easily to other components such as the network. If the network gets congested, users can be temporarily placed on broadcast channels until the congestion subsides. Such a scheme is vastly preferable to a scheme where content is dropped, resulting in a choppy presentation. A user's quality-of-service (QoS) is now measured by how often a user sees an advertisement in the event of failure or overload. A premium user would always have the desired bandwidth by paying more (except of course, in the presence of a catastrophic failure). This strategy is similar to the current CATV/DSS paradigm where users pay extra for subscribing to premium channels for the privilege of viewing programs with few interruptions.

The main focus of this paper is the use of rate adaptive stream merging and content insertion to provide overload tolerance. We design a mechanism that recovers the maximum amount of resources in a given amount of time with minimum impact to the clients and is fair to the clients. The rest of the paper is organized as follows. Section 2 provides the necessary background on resource reclamation techniques for continuous media servers and establishes the basis for our work. In Section 3 we develop a formal model of our system and then analyze recoverability under different overload scenarios. Section 4 is a discussion of the associated trade-offs. Section 5 summarizes the ideas presented.

2 Background

In this section, we describe a general architecture for a clustered CM server. We introduce the techniques of batching and clustering using rate adaptive stream merging and content insertion, all of which are related to our proposed scheme for overload recovery.

2.1 Server Architecture

Consider a general clustered video server architecture as shown in Figure 1. The system consists of several *servers* that disseminate content to *clients* via a delivery network. The delivery network consists of many *channels* that are accessible to all the clients.[2] This model is typical for most real world systems. Large-scale storage servers are built as monolithic units with large caches, inherent fault tolerance and are designed to meet high availability requirements. User transactions are handled by a cluster of front-end I/O processors that can access all the stored content.

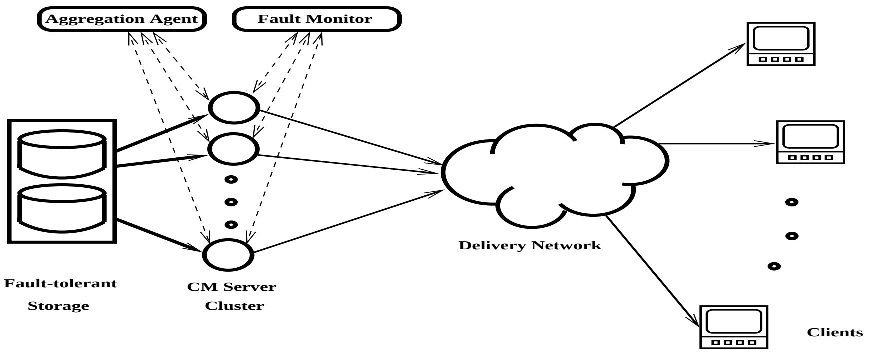

Figure 1: Server Architecture

Typically, any server can transmit on any channel but no two servers can use a particular channel at the same time and we assume the presence of a mechanism to ensure this mutual exclusion. It is in many ways similar to a CATV architecture but is general enough to accommodate a packet switched network in which channels can be implemented via multicast groups. Any channel in use would carry exactly one program *stream.* Furthermore, it is straightforward to implement some dedicated channels that continuously transmit advertisements, news clips or other programs of general interest. These channels permit content insertion in real-time and play an important role in the fault recovery scheme

[2]This assumption is only for illustration. Our scheme can be applied to other situations provided that a large number of clients and servers access a common channel set.

described in greater detail in Section 3. We now survey some of the schemes proposed in literature for resource reclamation in continuous media servers and discuss the applicability of these schemes for recovery from overload.

2.2 Batching

In *batching*, new playout requests for streams are grouped together at the time of request and channels are allocated to groups of users. With batching clients may have an initial waiting period which can cause them to renege their requests. Clients can be blocked due to all channels being consumed. The impact of channel allocation policies on quality of service in the CATV context is analyzed by Nussbaumer et al. [7]. Batching works well as long as the users do not interact. However, interactions cause users to break away from their groups defeating the initial gains. Such break-aways must be handled by starting a new independent stream for the user requesting interaction by drawing from a pool of *contingency channels*. A model for optimally allocating channels for batching, on-demand playback and contingency has been developed by Dan et al. [5].

When all contingency channels are consumed, any user interaction will block. This is an event which would occur with a very small but finite probability. This cannot be altogether avoided due to the statistical nature of the allocation policy. Periods of high interactivity can deplete all the free channels available and no more users can be admitted nor user interaction be permitted until some streams exit. This is a serious disadvantage of initial batching. Such situations can also be interpreted as a server overload in the context of this paper.

2.3 Merging

Rate adaptive stream merging (or adaptive piggy-backing) [6] is a technique that attempts to merge streams by varying their display rates. It has been observed that rate changes of $2-3\%$ by frame interpolation and expansion or contraction of the total length of the movie by up to 8% are acceptable in commercial video playback [6]. Content progression rates are distinct from the data delivery rate or the frame rate. An accelerated content progression rate implies that the total duration of the video will be reduced and any given scene would occur earlier in time.

Stream merging policies for resource reclamation in a healthy server are considered in [6]. Rate adaptive merging of streams reduces resource requirements both at the server and in the network. Batching can also be used in conjunction with merging for increased gains. We will now illustrate the process of rate adaptive merging with two streams carrying the same program [6]. .

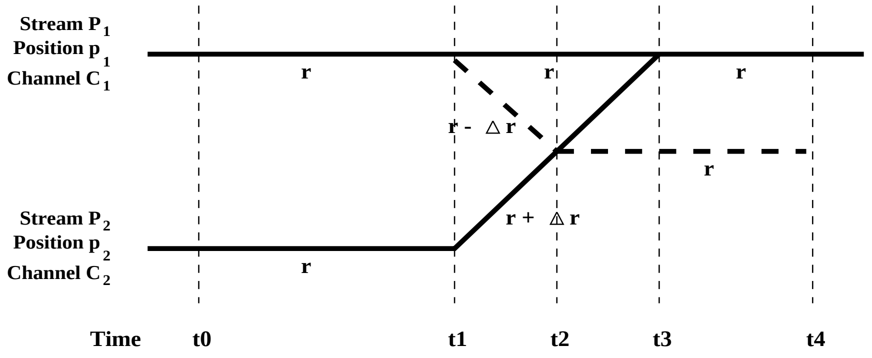

Figure 2: Rate Adaptive Merging of Streams

In Figure 2, at time $t = t_0$, the streams P_1 and P_2 carry the same program at rate r but with a temporal skew. Let the streams be at positions p_1 and p_2 respectively at time $t = t_1$. Let $p_2 < p_1$, and the positions are measured in terms of access units, like video frames for instance. By accelerating the content progression rate of stream P_2 by setting it to $r + \Delta r$ at time $t = t_1$, we can make the two streams reach the same point in the program at time $t = t_3$. The *catch-up window* is determined by the time interval :

$$T_c = t_3 - t_1 = \frac{(p_1 - p_2)}{\Delta r}. \tag{1}$$

The minimum duration of T_c is constrained by the maximum change allowed in the content progression rate. It is possible (see Section 4) to set the content progression rate of stream P_1 to $r - \Delta r$ at time $t = t_1$ and make the streams reach the same point in the program earlier, at time $t = t_2$ and the merging time interval $\hat{T}_c$ in this case is given by,

$$\hat{T}_c = \frac{(p_1 - p_2)}{2\Delta r}. \tag{2}$$

Depending on the rate adaptation policy, the streams reach an identical state at t_2 or t_3. At this point all clients receiving stream P_2 can be transferred to stream P_1 and the resources associated with P_2 can be released.

The size of the catch-up window is severely constrained by the maximum acceleration rate permissible from QoS considerations. If we wish to use merging in order to release channels in the event of failure, no resources are freed during the period of the catch-up window, which can be significantly long. If we wish to merge a failed stream, we allocate additional resources for it until it gets merged. It is likely that extra resources are unavailable in the event of failure and this shortfall can be significant when a large number of failed streams must be recovered. The next section elaborates on how content insertion techniques address this problem.

2.4 Content Insertion

Content insertion can be viewed as a coarse grained rate adaptation technique. This is the informational view – the primary content rate is altered by the introduction of secondary content. The secondary content can take the form of other programming. A different and operational view would explain content insertion as a form of *intermediate batching*. A third view arising from scheduling would explain the same technique as *time-sharing*. All these different views influence the application of content insertion in our proposed scheme.

Although television *commercials* are considered annoying by many, they subsidize the cost of entertainment provided to the consumer. With VOD services, these advertisements can lower per-user costs in an entirely new way by helping to diminish the number of concurrent streams via intermediate batching. Content insertion techniques are useful because they buy us time. This time can be used for time-sharing or in conjunction with merging to increase the catch-up window or alternatively, in the event of failure give a fallback allowing us to wait until resources become available. In the next section, we discuss why content insertion is critical to overload tolerance.

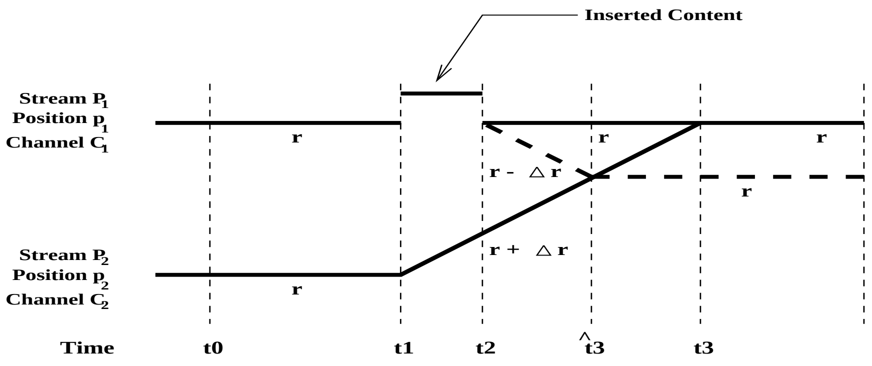

Figure 3: Modifying the Catch-Up Window through Content Insertion

Figure 3 illustrates how content insertion works in conjunction with merging to free resources during the catch-up window as well as to reduce T_c. Let us take the same case of two streams as in the previous section. By accelerating stream P_2 by setting its content progression rate to $r + \Delta r$ at time $t = t_1$, and by additionally inserting alternative content into stream P_1 from $t = t_1$ to $t_2 = t_1 + T_{ins}$ we can make the two streams reach the same point in the program at time $t = t_3$. T_{ins} should be less than the maximum acceptable duration, T_{ag} of inserted content at any one time. Let $p_3 = p_2 + T_{ins} * (r + \Delta r)$ be the position of stream P_2 at time t_2. The catch-up window is now given by

$$T_c = T_{ins} + \frac{(p_1 - p_3)}{\Delta r}. \tag{3}$$

It is possible to also set the content progression rate of stream P_1 to $r - \Delta r$ at time $t = t_2$ and make the streams reach the same point in the program earlier, at time $t = \hat{t}_3$ and the merging time interval $\hat{T}_c$ in this case is given by,

$$\hat{T}_c = T_{ins} + \frac{(p_1 - p_3)}{2\Delta r}. \tag{4}$$

However, T_{ins} cannot exceed T_c as this would lead to an oscillatory situation with the accelerated stream reaching a point in time ahead of the stream it is trying to catch up with. This yields the inequality :

$$T_c \geq T_{ins} \text{ or } p_1 \geq p_3.$$

If this condition is violated, then the inserted content must be stopped abruptly[3] in order to cluster the streams. Hence we can obtain the following bound on T_{ins}:

$$T_{ag} \leq T_{ins} \leq \frac{(p_1 - p_2)}{(r + \Delta r)}. \tag{5}$$

A similar bound can be achieved for $\hat{T}_c$, when the rate of both streams are being altered. This bound helps us to determine the feasibility of merging two streams using content insertion. If no insertion clip can be inserted into the stream in its entirety, the streams are allowed to merge, without any new content being inserted. This choice would be a policy decision, based on the system specifications.

Consider two streams that are spaced two minutes apart in a given program that must be clustered. This can be achieved by two minutes of content insertion into the leading stream and a channel is released in two minutes. With rate adaptive merging, clustering takes much longer. If we accelerate the trailing stream by 6.67%, [4] then the streams spaced two minutes

[3]This is undesirable if the inserted content consists of advertisements.

[4]This value corresponds to a tolerable limit for MPEG streams in a specific implementation by the authors.

apart would be clustered in 30 minutes. In addition, if we decelerated the leading stream, clustering occurs in 15 minutes. This is an unacceptably long time for overload recovery!

However, rate adaptive merging would still be required to fine tune the clustering process. Suppose that the inserted content is in the form of 30 second clips and there are several content insertion channels such that a clip starts on some content insertion channel every 5 seconds. Content insertion can be done smoothly only if the temporal skew between streams are exactly integral multiples of 5 seconds and the failure or interaction that triggers clustering is aligned with the start of an insertion clip, both of which are unlikely to happen. However, these small fractional skews from the start of the clips can be eliminated by the use of rate adaptation.

Content insertion can be applied to overload situations in an entirely different way as well. Consider a server that has a capacity to serve $1,000$ channels and the system demand requires the existence of $1,025$ streams. If we ignore merging and focus on content insertion alone, assuming that we have content insertion channels, we could switch 25 streams on to these channels for 60 seconds. Streams can be vacated in rotation and the cycle would complete in 40 minutes. Thus streams would receive an *ad-dosage* of 1 minute once every 40 minutes. In the event that continuous resource reclamation is carried out using merging, the situation would improve vastly. If the contrary is true, and the overload increases, it would result in degradation of service manifested as an increased amount of inserted content. We refer to this situation as *stream-thrashing*. Arguments similar to overload in time-shared operating systems come to bear – there are always intrinsic checks and balances that would reduce popularity with service degradation on overload.[5] Section 4 describes additional advantages due to content insertion.

3 Proposed Scheme for Overload Tolerance

In this section, we formalize our model based on the concepts introduced in Section 2 and apply it to analyze recoverability under failure and propose schemes for recovery under different overload scenarios. Table 1 describes the notation used in the paper.

[5]Detailed analysis of the effectiveness of these heuristics and fairness issues are beyond the scope of this paper.

3.1 Formal Model of the System

Let the total number of servers in the cluster be N and the servers are denoted by S_i ; $i \in \{1, 2, ..., N\}$. Let $\mathcal{S}$ denote the cluster. To simplify analysis, we assume that all streams have identical bandwidth requirements. However, the maximum bandwidth of each server can be different in the case of a heterogeneous server cluster. Without loss of generality we can use the bandwidth of a single stream as a unit of server bandwidth.

Let B_i be the total bandwidth of the server S_i in terms of the number of streams that the server can support. Let $L_{i,t}$ be the load on server S_i at instant t, in number of streams. The total bandwidth $\mathcal{B}$ available in the system is given by $\mathcal{B} = \sum_{i=1}^{N} B_i$. The total load $\mathcal{L}_t$ of the system at instant t, is given by $\mathcal{L}_t = \sum_{i=1}^{N} L_{i,t}$. During normal operation, $\mathcal{L}_t \leq \mathcal{B}$ and violating this constraint constitutes an overload.

The interplay between various system components is described via an entity-relationship diagram (Figure 4). A program stream plays out on a given channel which can be accessed by several clients. Some channels can be idle and some active channels carrying alternative content may have no clients listening. A given stream is serviced by one server on any one channel. Clients can switch between channels autonomously or can be mandated due to clustering. The special case which is of interest to us, occurs when the clustering is forced due to the occurrence of faults. Similarly, streams can be terminated or reassigned between servers on clustering, on fault or possibly to achieve load balancing.

Let $\mathcal{U}_t = \{U_j\}$ be the set of clients in the system at time t and $\mathcal{C} = \{C_k\}$ be the set of channels in the system. Let the set of active streams in the system at time t be $\mathcal{P}_t = \{P_l\}$. If channels are reclaimed continuously by merging, then two streams will carry the same program if and only if there is a temporal skew between them. The total number of clients $|\mathcal{U}_t|$ can be greater than or equal to the number of streams $|\mathcal{P}_t|$. Typically $|\mathcal{P}_t| \leq |\mathcal{C}|$, and violation of this constraint constitutes an overload. $|\mathcal{U}_t|$ can exceed $|\mathcal{P}_t|$ by virtue of clustering. The global state of the system at any given time t is given by the following maps :

- Channel Assignment : The channel assignment function, $c_t(P_i)$ is injective and assigns each stream to a channel and a server. It is given by $c_t : \mathcal{P}_t \rightarrow \mathcal{C} \times \mathcal{S}$.
- Stream State : The state of the stream is given by $P_l \mapsto (p_l, s)$ where$P_l \in \mathcal{P}_t$. $p_l \in Z^+$ is the position of the stream P_l within the program and the state of the session $s \in \{$ normal playback, accelerated playback, fast forward, rewind, paused, vacated $\}$.

- Client Assignment : The client assignment function $u_t : \mathcal{U}_t \rightarrow \mathcal{P}_t$ assigns a client to a given stream.

We also require that the following fundamental primitives, that enable the server to control the client, are available. They are necessary in order to implement the functions of merging and content insertion.

```
ReassignStream(StreamId, CurrentServerId, NewServerId)
SwitchChannels({ClientId}, CurrentChannelId, NewChannelId)
```

Table 1: List of Symbols Used

Symbol	Description
$\mathcal{B}$	total system bandwidth in number of streams
$\mathcal{B}^f$	loss of system bandwidth due to failure
B_i	the bandwidth in number of streams of the i^{th} server
$\mathcal{C}$	set of channels in the system
C_k	the k^{th} channel
$\mathcal{L}_t$	total system load in number of streams at time t
c_t	channel assignment function
$g(U_j)$	penalty function for the j^{th} client
$L_{i,t}$	the load in number of streams on the i^{th} server at time t
N	number of servers in the cluster
$\mathcal{P}_t$	set of streams in the system at time t
P_l	the l^{th} stream
p_i	position in program for the i^{th} stream
r	content progression rate in frames/second
$\mathcal{S}$	set of servers in the cluster
S_i	the i^{th} server
T_c	catch-up window time between streams
T_{ins}	duration of inserted content for a stream at any one time
T_{ag}	duration of each content insertion clip
T_{rec}	time to recover from overload
T_{cycle}	time for one cycle of vacations of streams
$T_{adspacing}$	minimum time between two content insertion periods
T_{conf}	time to reconfigure a spare server
$\mathcal{U}_t$	set of clients in the system at time t
U_j	the j^{th} client
u_l	client assignment function

Let λ_a and μ_d respectively denote the mean rate of generation of new streams due to program requests and the average number of streams that leave the system due to program termination in the system. Let μ_m be the mean merging rate, which is the average number of streams that get merged across all servers in unit time and let λ_f be the mean forking rate, which is the average number of streams that break away due to client interaction across all servers. Thus μ, the rate at which channels are released in the system, is given by $\mu = \mu_m + \mu_d$, and λ, the rate at which channels are consumed in the system, is given by $\lambda = \lambda_f + \lambda_a$. The stability criterion requires that $\lambda \leq \mu$. This is the steady state referred to later, in Section 3.3.

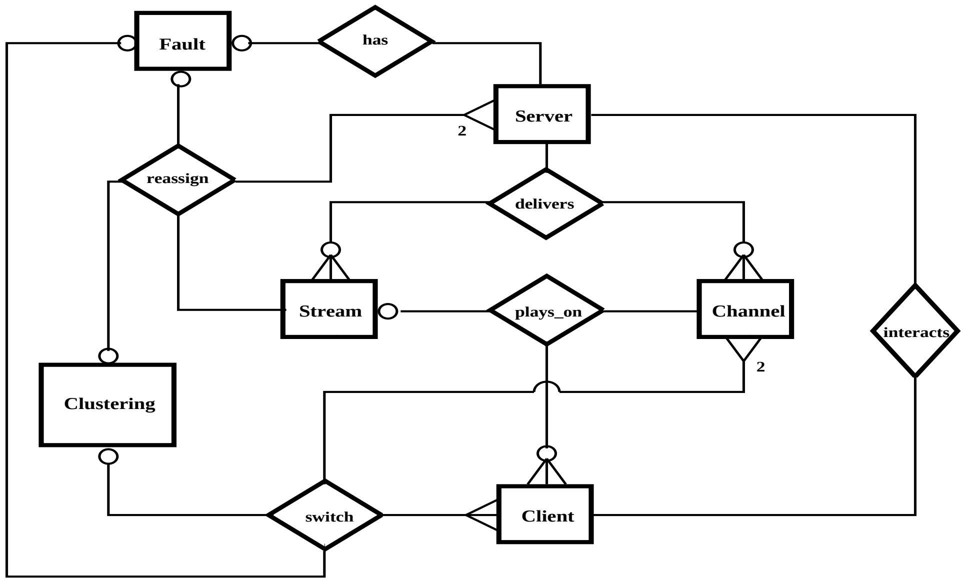

Figure 4: Entity-Relationship Diagram of the System

3.2 Failure Model and Recoverability

In Section 2, we surveyed the basic techniques for resource reclamation. In this section, we develop a fault model that enables us to apply the techniques of merging and content insertion to recover from the overload arising due to failures. We then introduce our proposed procedure to handle different levels of overload. In the context of the server architecture described in Figure 1, the fault monitor and storage are also assumed to be inherently fault-tolerant. This is a reasonable assumption when storage is an independent component. For enhancing fault tolerance, the stream fault monitor which checks for the health of all streams must be implemented as an independent component. However it is possible to implement the monitoring function distributed among the servers with some loss of reliability.

Let us assume that at most q out of N servers in the cluster can fail simultaneously and

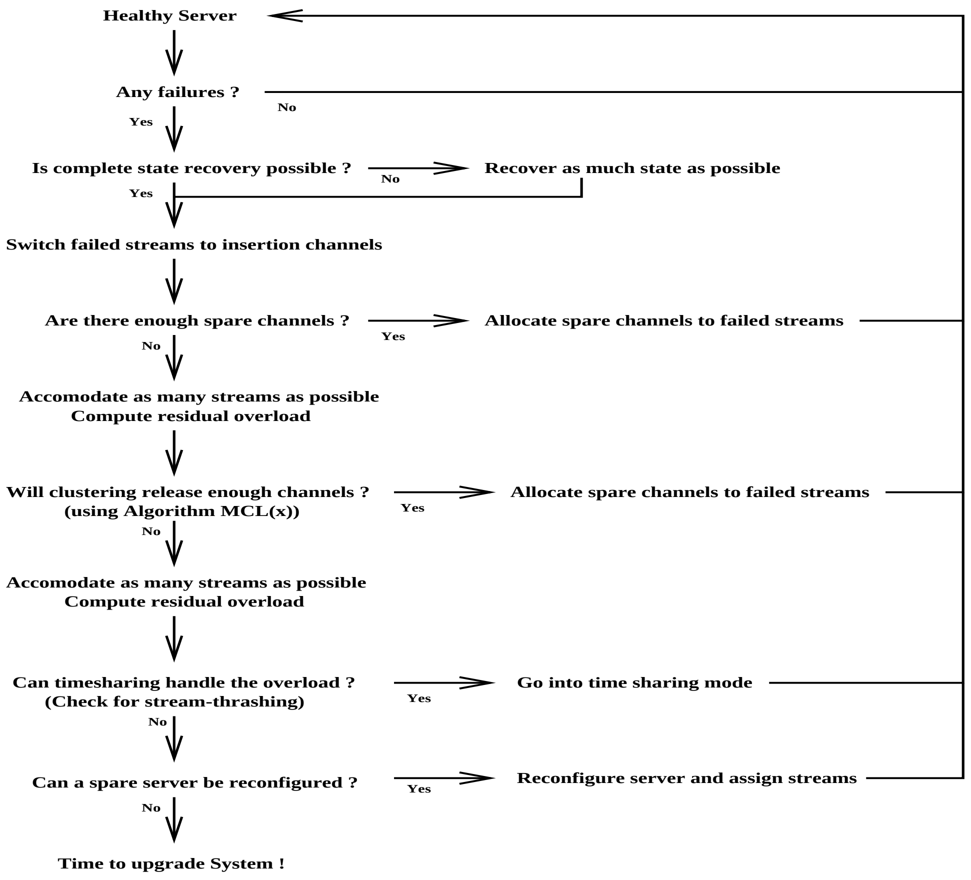
Healthy Server
Any failures ?
No
Yes
Is complete state recovery possible ?
No
Recover as much state as possible
Yes
Switch failed streams to insertion channels
Are there enough spare channels ?
Yes
Allocate spare channels to failed streams
No
Accomodate as many streams as possible
Compute residual overload
Will clustering release enough channels ?
(using Algorithm MCL(x))
Yes
Allocate spare channels to failed streams
No
Accomodate as many streams as possible
Compute residual overload
Can timesharing handle the overload ?
(Check for stream-thrashing)
Yes
Go into time sharing mode
No
Can a spare server be reconfigured ?
Yes
Reconfigure server and assign streams
No
Time to upgrade System !

Figure 5: Overload Recovery Procedure

that no additional failures occur during the recovery period.[6] Restoring service to the failed streams in the event of a failure requires the following:

- The state of each individual streams at the point of failure. We need to know the state of the streams that were initially served by the q servers that failed. This can be achieved in several ways:
  - The state of any server is stored in at least $q+1$ servers. This ensures state recovery on the event of failure of at most q nodes. Alternatively a more sophisticated error correcting code can be used to stripe the state information with some performance overhead.
  - Global state is stored in the monitor and each server knows its own state. When the monitor fails independently, backup copies of the monitor must take over.
  - Recovery from failure could be initiated by the client. Stream states on each channel could be retrieved by multicasting requests on each channels that must be restored. Resolution of duplicate replies can be performed by taking the first reply or by voting. However client initiated recovery implies temporary disruption of service at client. This justifies the use of the fault monitor at the server.
- A fixed number of channels that broadcast advertisements or other alternative content continuously are assumed to be continuously available. In the event of a server failure, the failed streams are transferred to these channels before recovery. This is necessary because :
  - There is a significant initial latency involved in starting up a new stream even if there is sufficient spare bandwidth [1].
  - Overload due to simultaneous failure of a large number of streams would involve significant computation time for resource allocation.
  - Resources can be unavailable and the overload cannot be handled until some streams are merged. Merging of streams involves a finite catch-up window as described in Section 2.4.
  - Resources can be unavailable even in the long-term and some streams must be dropped. From a point of view of revenue, there is more incentive in retaining popular program streams since they would on average, allow more clustering in

[6]This assumption translates to a loss of a fraction of the available bandwidth for serving streams.

the future. The system requires a finite computation time to determine which streams to drop.

- The content insertion channels can be used to implement a time-sharing mechanism that cyclically transfers clients to content insertion channels to handle transient overloads or while the system configures a new server to handle the overload.

The procedure for handling overload on failure is shown in Figure 5. We consider several overload scenarios and show how we can handle situations not only when there are more clients than channels, but also when there are more streams than are channels! The first step of the overload recovery procedure involves restoration of stream states. On failure of servers, the failed streams are vacated to content insertion channels as a reflex response. As a first measure, if there are spare channels available, they are allocated to as many streams as possible. The next attempt is to recover channels by clustering. This is solved by the algorithm proposed in the next section which determines the maximum number of clusterings possible within a given amount of time. If the channels released are insufficient to handle the overload, the overload is handled by time-sharing of the channels. If the time-sharing approach is not viable, we configure additional servers to handle the overload. If there is an overload beyond this, the failure is considered catastrophic and streams must be dropped according to a suitable pricing policy (not considered in the paper.)

Given that there is a fault monitor in place and content insertion channels are available, we analyze the conditions that permit partial or complete recovery from the overload, in the next section. The bandwidth consumed by the content insertion channels is negligible and is not considered in the analysis. Also, content insertion channels are distributed across all servers and some of them are available even during failure conditions. We also consider *soft-failure* when there is no server failure but there is a transient overload resulting from increased interactivity. A solution based on *time-sharing* of the channels via content insertion into streams in a round-robin fashion is proposed to handle transient overload.

3.3 Recoverability Analysis

Let us assume that the system is in the steady state before the fault occurs at time $t = t_0$. Let $k \in \{1, ..., q\}$; $q < N$, be the number of servers that fail simultaneously where q is a design parameter of the system. Let $\mathcal{F} = \{S_i | S_i\ is\ a\ failed\ server\}$. Clearly $|\mathcal{F}| = k$. This failure generates an extra load $\mathcal{L}^f_{t_0}$ that must be accommodated, given by $\mathcal{L}^f_{t_0} = \sum_{\{i|S_i \in \mathcal{F}\}} L_{i,t_0}$. The loss of bandwidth in the system is $\mathcal{B}^f = \sum_{\{i|S_i \in \mathcal{F}\}} B_i$. Thus the bandwidth available after failure is $\mathcal{B}' = \mathcal{B} - \mathcal{B}^f$.

Case 1 :

Complete recovery is guaranteed for all streams if net overload is lower than spare bandwidth after failure. This is the trivial case, $\mathcal{L}_{t_0} < \mathcal{B}'$, when the system has spare bandwidth to accommodate the extra load due to failure.

Let T_r be the computation time required for resource allocation and T_s be the startup latency associated with starting up a stream. We assume that these are small finite constants in the order of a few seconds. The failed streams will have or content insertion for a period $T_{rec} = T_r + T_s$ seconds. This is also the recovery period for the entire system. This represents the baseline user penalty on failure. If there is not enough bandwidth available to reassign all the failed streams to new channels, then a procedure to recover from the residual overload is discussed in Case 2.

Case 2 :

Complete recovery is guaranteed if total load after clustering is lower than the total bandwidth after failure. Clustering of streams involves both content insertion and rate adaptive merging. Content insertion is more powerful and is a coarse grained technique as described in Section 2.4.

We now formulate the clustering problem that handles the overload scenario of Case 2. Suppose the system can support X channels after failure, and there are x failed streams that have been switched to content insertion channels. Without loss of generality we can assume that that all X channels are currently in use and therefore, we need to release x within a maximum time to recover, T_{rec}. If all X channels were not in use, we can assign the available free channels to as many failed streams as in Case 1 and thereby reduce it to Case 2. The time to recover, T_{rec} represents the maximum content insertion period for clients.

The solution to the problem first involves determining whether it is possible to release x channels within T_{rec} and if not, what is the maximum number of channels that can be released by clustering. The solution must also include the list of streams that are to be clustered together. If less than x channels can be released by clustering, we release as many as possible and handle the net overload after clustering by a different mechanism using time-sharing channels. This would be the Case 3 scenario explained later.

We now propose an algorithm called EMCL(x) which takes the number of channels, x, that are required to be released as an argument and returns a list of clusters to release x or more channels. If x channels cannot be released, then EMCL(x) returns a list of clusters which release the maximum number of channels possible. We claim and prove that EMCL is correct and it executes in polynomial time.

Definition 1 *Given two streams P_i and P_j, we define the distance between them $d(P_i, P_j)$ as the time needed to cluster P_i and P_j to release a channel. If P_i and P_j carry different programs then $d(P_i, P_j) = \infty$.*

Definition 2 *A cluster, denoted by $(P_i, P_j), P_i \neq P_j$ is a group of streams carrying the same program such that every stream in the cluster has a program-position in between and including that of streams P_i and P_j and such that $d(P_i, P_j) \leq T_{rec}$. The size of the cluster $|(P_i, P_j)|$ is the number of streams in the cluster and at the end of clustering, $|(P_i, P_j)| - 1$ channels would be released.*

Definition 3 *The cluster (P_i, P_j) is the earliest in a given program if there is no other cluster (P_m, P_n) such that P_m has a program-position earlier than that of P_i.*

Definition 4 *A cluster (P_i, P_j) is a maximum cluster for a given program, if $|(P_i, P_j)|$ is maximum for that program.*

Definition 5 *A cluster (P_i, P_j) is maximal if there exists no (P_i, P_k) such that $d(P_i, P_j) < d(P_i, P_k) \leq T_{rec}$. It is the set of all streams leading P_i that P_i can be clustered with, in time T_{rec}.*

Algorithm EMCL(x)

```
1. $M \leftarrow \{\}$
2. $k \leftarrow 0$
3. $S \leftarrow \{(P_i, P_j) \mid \forall P_i \in \mathcal{P}_t,\ (P_i, P_j)$ is a maximal cluster$\}$
4. $M \leftarrow M \cup \{(P_p, P_q) \in S\}$ where $p_p$ is min. for the program and $|(P_p, P_q)|$ is max. across programs
5. $k \leftarrow k + |(P_p, P_q)| - 1$
```

```
6. if $k \geq x$ return $M$
7. $S \leftarrow S - \{(P_p, P_q)\} \cup \{(P_r, P_s) \mid p_p < p_r \leq p_q\}$
8. if $S = \{\}$ return $M$
9. goto 4
```

For the sake of analysis we assume that no two streams considered for clustering are in identical positions in a program. The EMCL algorithm is quite straightforward. It picks the earliest maximal cluster which is largest across all movies and adds it to the list of clusters. This process is continued until we have picked as many or more than the requested number of streams to be merged, or until no more streams can be clustered. The largest cluster across all movies is chosen so that the loop terminates earlier when only a small number of channels are required.

Theorem 1 *(Complexity) EMCL(x) runs in polynomial time.*

Proof : Suppose every stream in $\mathcal{P}_t$ is indexed numerically and the position of each stream in the program it carries is also known. We can sort the list of streams by the program index and then by increasing order of positions. Using a counting sort, this would take $\mathcal{O}(n)$ time in terms of the number of streams, n, in the system. The set S in line 3 can then be constructed in $\mathcal{O}(n^2)$ time. Lines 4, 5 and 7 take $\mathcal{O}(n)$ time while lines 1, 2, 6, 8 and 9 take $\mathcal{O}(1)$ time. Since the loop in lines 4 – 9 might be performed at most $n/2$ times which is the maximum possible number of clusters with n streams. Therefore this loop has an overall complexity of $\mathcal{O}(n^2)$. Thus EMCL(x) has a polynomial time asymptotic complexity, $\mathcal{O}(n^2)$. $\square$

Theorem 2 *(Correctness) EMCL(x) returns a list that gives at least x clusterings whenever such a set exists. Otherwise EMCL(x) returns a list which corresponds to the maximum number of clusterings that are possible.*

Proof : Suppose there are n streams in the system and at most α clusterings are possible. Assume that EMCL(α) returned only β clusterings where $\beta < \alpha$. $\Rightarrow \exists$ at least one stream P_p which could have been clustered with some other stream P_q but was not returned by EMCL(α). Let P_p be the first such stream in some program. If P_p was the very first stream carrying that program, then (P_p, P_q) is part of a maximal cluster. Also, since P_p is the very first stream in the program, it would be part of the earliest maximal cluster which would have been picked up before the loop in lines 4 – 9 of the algorithm terminated. However since it was not returned, *it implies that no such P_q exists, which contradicts the assumption that P_p could have been clustered.*

If P_p is also the last stream in the program, any stream P_q with which it could have been clustered is already part of another maximal cluster. Since this cluster is maximal it could not have included P_p. Therefore, if P_q were clustered with P_p, then it could not have been clustered with its present group. So the number of clusterings would remain the same, either way. This means β cannot be lesser than α.

If P_p was somewhere in the middle and suppose there existed a stream P_q with which it could be clustered. If P_q was earlier than P_p, we can apply an argument similar to when P_p was the last stream to prove that β cannot be lesser than α. If P_q was later than P_p, then (P_p, P_q) would have been part of an earliest maximal cluster and would have been chosen. This contradicts our assumption. Thus EMCL(α) would return α clusterings if that was the maximum number possible.

If asked for more than the maximum, the algorithm will still return the maximum by virtue of line 8. If asked for less than the maximum, it may return a list that results in more clusterings than requested, since the clusters chosen may not add up exactly to what is requested. Thus EMCL(x) is correct. $\square$

Choosing the earliest maximal cluster is key to our algorithm's correctness. Greedy algorithms which cluster nearest streams pair-wise [6] or those which pick the maximum cluster first can easily be shown to be sub-optimal by trivial counter-examples. Choosing the earliest cluster also offers a distinct advantage. The algorithm would work equally well if we defined maximal clusters around group leaders in the inverse direction and pick the latest maximal cluster instead. However there is little advantage to be gained by clustering streams which are going to terminate soon anyway. By clustering streams that are in the earlier part of the program, the gains are valid for a longer period, provided there are not many break-aways due to interactions.

Though it is possible to refine the algorithm further to improve performance and to check for violation of per-user constraints, these are not relevant in the context of this presentation where we mainly wish to show that we can compute in reasonable time the extent of recovery from overload that can be achieved via clustering. We present another algorithm MCL(x), which is a refinement of EMCL(x), that runs in $\mathcal{O}(n)$ time and lends to direct implementation. MCL(x) differs from EMCL(x) in that it does not necessarily pick the earliest cluster first. The complexity analysis is trivial and correctness of this algorithm follows from Theorem 3.

Algorithm MCL(x)

```
1. $M \leftarrow \{\}$
2. $S \leftarrow \{\}$
3. $k \leftarrow 0$
4. $L \leftarrow \mathcal{P}_t$
5. sort $L$ by program indices and then by increasing order of positions
6. let $\leq$ represent the overall order in $L$
7. while $\exists (P_i, P_j) \in L$
8.         let $(P_p, P_q)$ be the first maximal cluster in $L$
9.         $S \leftarrow S \cup \{((P_p, P_q), |(P_p, P_q)|)\}$
10.        $L \leftarrow L - \{P_k \in L \mid p_k \leq p_q\}$
11. sort $S$ by decreasing order of cluster sizes
12. while $\exists ((P_i, P_j), |(P_i, P_j)|) \in S$
13.        $M \leftarrow M \cup \{(P_p, P_q)\}$ where $|(P_p, P_q)|$ is a max. cluster in $S$
14.         In the previous line, we choose one arbitrarily if the max. is not unique
15.        $S \leftarrow S - \{((P_p, P_q), |(P_p, P_q)|)\}$
16.        $k \leftarrow k + |(P_p, P_q)| - 1$
17.        if $k \geq x$ return $M$
18. return $M$
```

Theorem 3 *The set S at line 11 in MCL(x) corresponds to the maximum possible number of clusterings.*

Proof: Omitted for brevity. $\square$

To simplify analysis, we neglected the possibility of interaction during clustering. However this does not affect the performance or correctness of our algorithm. Suppose γ interactions occur during clustering, we can handle them by vacating γ arbitrary streams in the system that are not being clustered currently and handle the interactions with these channels. The vacated streams are transferred on to content insertion channels. We also invoke EMCL(γ) to release γ channels to restore the channels that were vacated. If however, γ channels are not available this represents an overload condition that cannot be handled by clustering and this condition is handled in Case 3 by means of time-sharing available channels.

Case 3 :

If total load after clustering is marginally higher than the total bandwidth after failure, we can increase the *virtual bandwidth* of the system by switching clients to content insertion channels in rotation. This is basically time sharing of available channels. The situation is analogous to a multiprocessor system where the number of processes slightly exceeds the number of processors and therefore the percentage of time that each process is idle is small and the interval between idle periods is large. With this technique, there is a risk of stream thrashing, that is, when clients are vacated to content insertion channels very often.

In the same way that processes spend more time swapping than in computation when an operating system thrashes, during stream thrashing the users "feel" that they are getting more inserted content than the actual program being watched. Therefore the Case 3 overload problem is one in which we determine if it is possible to handle the overload by time sharing. This technique is also suited to handle transient overload arising when a lot of clients interact simultaneously.

When used in conjunction with Case 2, all streams that are currently being clustered should not be part of time-sharing. Suppose the system can support X channels over and above those which are being clustered. Let x streams be the residual overload which means we need a virtual bandwidth of $X+x$ streams out of a real bandwidth of X streams. Suppose $T_{adspacing}$ is the minimum time interval between two content insertion periods for any user. Let us also neglect prior content insertion if any. Time sharing involves vacating x streams in rotation and the cycle completes approximately in time, $T_{cycle} = (X + x)/x$. Overload can be handled without stream thrashing provided $T_{cycle} \leq T_{adspacing}$. If this is violated, we have the scenario outlined in Case 4.

Case 4 :

If we determine that stream-thrashing is imminent, by applying the analysis in Case 3, it is still possible to recover from the overload gracefully, provided we can reconfigure spare servers into the system to handle the overload before stream-thrashing becomes apparent. If the time to reconfigure a spare server is T_{conf} the system can recover gracefully if $T_{conf} \leq T_{cycle}$. If spare servers cannot be configured, the overload cannot be absorbed completely and some streams must be dropped.

4 Discussion

If an interval caching policy [4] is implemented on each server, the stream allocation strategy might try to optimize cache usage by placing mergeable streams on the same server. Unfortunately these streams share the same fate in the event of server failure. This reduces the possibility of channel reclamation and therefore overload recovery. This situation can be avoided by allocating streams that can be clustered across different servers. While this may be suboptimal from the caching perspective, it does not affect gains from clustering and will improve overload tolerance. As an alternative to content insertion channels, clips for insertion could be stored entirely in the cache.

In our paper live transmissions are irrelevant for the following reason. We need only one channel for each live broadcast. Streams which are at any temporal skew from the live broadcast are not “live” by definition. Assuming content is being spooled, replays would be permissible within the content that has been spooled. This spooled content represents just another program in the system for all practical purposes.

Although with rate adaptive merging, it is possible to decelerate the rate of the leading stream, it is not practical when there are a large number of trailing streams that we want to cluster with it. Also, practical considerations favor that we handle only the normal and accelerated content progression rates. In the introduction we stated that merging involves processing overheads. A practical solution for supporting rate adaptive merging of streams from content stored in a single format with negligible processing overhead for MPEG encoded streams is proposed elsewhere. We can therefore justify the use of merging to perform fine-grained clustering.

Clearly, content insertion improves on merging as resources are freed earlier than merging as described in Section 2.4. The other advantage is the use of ad revenues to offset the cost to the user. In this case the ads are also subsidizing costs in an entirely new way by permitting aggregation. A third advantage is that the technique can be applied to schemes other than merging. Content insertion techniques can also be applied to handle transient overload phenomena. Interactions cause clients to break-away from their groups, defeating the gains that are made via batching [5]. Since each break-away claims a new channel, with a finite probability, the system can be depleted of all channels. This causes future interactions to block. The clustering algorithm described can be used to recover resources. Some of the criteria that can be used to determine the events that trigger the clustering algorithm include :

- A high degree of interaction in the whole system.
- A high degree of interaction in a given movie. This is a refinement to the above as interactions in less popular movies do not trigger the algorithm.
- The existence of several streams for the same movie. This is a bad choice for a trigger because there can be several streams running at large skews that cannot be merged.
- The occurrence of an interaction block. However an aggressive scheme will prefer a preventive approach over this reactive approach.

Clustering involves degradation of quality of service to the customer and it is necessary that the clustering policy is fair to all the customers. Therefore, the content-insertion policy must ensure that each customer receives no more than the maximum allowed advertisement time (*ad-dosage*) and no less than a minimum ad-dosage so that advertising revenue objectives can be satisfied. Customers should get an equitable distribution of the total ad-dosage. Most importantly, customers should experience no less than a specified minimum period between advertisements. If this is violated on average, then it is an indication of *stream-thrashing* in the system. Finally we note that it is reasonable to implement an overload recovery scheme that avoids Case 2 in the interest of simplicity and handles overload directly using Case 3 and Case 4. However, Case 2 is attractive as it can handle a higher degree of overload and represents a potential mechanism to increase system utilization during peak loads.

Since merging and content insertion result in service degradation, we define a user penalty metric which is composed of three different components : the duration of inserted content that the user must tolerate, the duration that content is delivered at an altered rate, and the interval between two content insertion periods. Rate adaptation can be done without significant loss of perceived quality while content insertion is clearly obvious. Users are sensitive to the frequency of content insertion and it is desirable to space them as widely apart as possible. Therefore the components are weighted differently to reflect this fact.

The penalty on the overall system is the average penalty for all users in the system and the objective is to minimize this quantity while trying to maximize the number of streams supported. Fairness considerations require that the variance in user penalty among users be as low as possible. We can ensure that constraints for all users are met by pessimistically scheduling the constraints. In other words, we can make aggregation and vacation decision on groups based on constraints corresponding to the most limiting users in the group.

The penalty function is given by $g : |\mathcal{U}_t| \rightarrow Z_0^+$, where Z_0^+ is the set of non-negative integers. We assume a digital system in which time is counted in discrete steps. For the given user U_j, let $T_{ins,i}$, $T_{mer,j}$ and $T_{gap,k}$ respectively denote the duration of the i^{th} content insertion, the duration of the j^{th} merging interval and the gap between the k^{th} and $k+1^{th}$ content insertions. The penalty function is then defined by,

$$g(U_j) = c_0 * \sum_i T_{ins,i} + c_1 * \sum_j T_{mer,j} + c_2 * \sum_k (1/T_{gap,k}). \qquad (6)$$

Server performance is indicated by the maximum number of clients that can be supported by the system for a given average client penalty averaged over all clients in the system. The objective is to minimize $\Sigma_t(\sum_{U_j \in \mathcal{U}_t} g(U_j)/|\mathcal{U}_t|)$.

Other useful indicators are the time to recover from overload and the *packing density* which is the average of the ratio of the number of clients to the number of streams. Since scheduling is cyclic in most continuous media servers [1], each data point is a single scheduling cycle. A high packing density means that we can increase the number of clients and hence the revenue. On the negative side, it also means a higher chance of transient overload. It is possible to honor per-user penalty by making each group adopt the constraints of the most limiting user in the group. However this may not provide the globally optimal system utilization. In an earlier section, we analyzed the extent of overload recovery that can be achieved within a given maximum time interval. Some per-user constraints may be violated during overload since the recovery algorithms do not evaluate the user penalty function. However we can pre-filter the streams for which constraints may get violated and take them out of the scope of the recovery algorithms.

5 Summary

Overload recovery in CM servers differs from that in conventional servers due to the requirement of continuity in playout. In this paper we showed that clustering by merging and content insertion can be applied to provide graceful recovery and overload tolerance on failure in CM servers. We proposed a procedure which can handle different degrees of overload efficiently and in a practical manner. We also provide a linear time algorithm to determine whether clustering can release the required amount of bandwidth. We also showed how classical ideas of time-sharing in multiprocessor systems can be applied to handle transient overload phenomena. We have presented a linear time algorithm which determines the maximum number of clusterings possible within a fixed recovery period.

We have discussed how content insertion when applied to resource reclamation and overload recovery can subsidize user costs in an entirely novel and significant way. While the domain of discourse in the paper is overload scenarios in clustered CM servers, the techniques proposed can easily be adapted to other domains such as congestion or failure in the network.

References

[1] D. Anderson, Y. Osawa and R. Govindan, “A File System for Continuous Media,” *ACM Transactions on Computer Systems,* Vol. 10, No. 4, November 1992, pp. 311-337.

[2] S. Berson, L. Golubchik and R. R. Muntz, ”Fault Tolerant Design of Multimedia Servers,” *ACM SIGMOD International Conference on Management of Data,* San Jose, CA, USA, May 1995, pp. 364-375.

[3] T. Chiueh and R. H. Katz, “Multi-Resolution Video Representation for Parallel Disk Arrays,” *Proceedings of ACM Multimedia’93,* Anaheim, CA, USA, August 1993, pp. 401-409.

[4] A. Dan, D. M. Dias, R. Mukherjee and D. Sitaram, R. Tewari, “Buffering and Caching in Large-Scale Video Servers,” *Compcon 95*, San Francisco, CA, USA, March 1995, pp. 217-224.

[5] A. Dan, P. Shahabuddin, D. Sitaram and D. Towsley, “Channel Allocation under Batching and VCR Control in Video-On-Demand Systems,” *Journal of Parallel and Distributed Computing (Special Issue on Multimedia Processing and Technology),* Vol. 30, No. 2, November 1995, pp. 168-179.

[6] L. Golubchik, J. C. S. Lui and R. R. Muntz, “Reducing I/O Demand in Video-On-Demand Storage Servers,” *SIGMETRICS ’95/Performance ’95 Proceedings,* Ottawa, Canada, May 1995, pp. 25-36.

[7] J-. P. Nussbaumer, F. Schaffa, "Impact of Channel Allocation Policies on Quality of Service of Video on Demand over CATV," *Multimedia Tools and Applications,* Vol. 2 , Kluwer Academic Publishers, 1996, pp. 111-131.

[8] B. Ozden, R. Rastogi, P. Shenoy and A. Silberschatz, "Fault-tolerant Architectures for Continuous Media Servers," *ACM SIGMOD International Conference on the Management of Data,* Montreal, Canada, June 1996, pp. 79-90.

[9] R. Tewari, D. Dias, R. Mukherjee and H. Vin, "High Availability for Clustered Multimedia Servers," *Proceedings of International Conference on Data Engineering,* New Orleans, USA, February 1996.

[10] H. M. Vin, P. J. Shenoy and S. Rao, "Efficient Failure Recovery in Multi-Disk Multimedia Servers," *Proceedings of the Twenty-Fifth Fault Tolerant Computing Symposium,* Pasadena, CA, June 1995, pp. 12-21.