Multimedia Tools and Applications 8, 219–247 (1999)

A Distributed Fault-Tolerant Design for Multiple-Server VOD Systems

ING-JYE SHYU ejshue@csie.nctu.edu.tw
SHIUH-PYNG SHIEH ssp@csie.nctu.edu.tw
Department of Computer Science and Information Engineering, National Chiao-Tung University, Hsinchu, Taiwan, R.O.C.

Abstract. Fault tolerance is an important design criterion for reliable and robust video-on-demand systems. Conventional fault-tolerant designs use either a primary backup or an active replication method to provide system fault tolerance. However, these approaches suffer from low utilization of the backup or replication system. In this paper we propose two playback-recovery schemes for distributed video-on-demand systems called the *forward playback-recovery scheme* and the *backward playback-recovery scheme*. Unlike conventional fault-tolerant designs, our schemes use existing playback resources to recover faulty playbacks without allocating new resources, significantly reducing recovery overhead. To use the schemes effectively, we developed a distributed algorithm for determining the order and gap information between the playbacks on the distributed video-on-demand servers so that overhead for recovering from a server failure can be minimized. This algorithm achieves $N-1$ fault-tolerant resiliency for N-server video-on-demand systems. In addition, three server-recovery policies are also presented to guide surviving servers in applying the proper scheme to recover faulty playbacks, thus reducing overall recovery costs. Simulation results show that the proposed recovery schemes are effective and useful in designing fault-tolerant multiple-server video-on-demand systems.

Keywords: fault tolerance, fault recovery, distributed algorithms, multimedia systems

1. Introduction

Video-on-demand (VOD) applications have recently received much attention from the telecommunications, entertainment and computer industries [6, 7, 9, 12]. And in essential or commercial VOD applications, fault tolerance is one of the most important issues because it allows systems to accommodate component failures [11, 16]. The most common approach to providing fault tolerance uses redundancy, organizing the redundant components as either *active replication* or *primary backup* units [3, 15, 19]. In a dual-server system with an active replication scheme, both servers work in parallel in order to accommodate failures. When the primary server fails, the secondary server accepts the workload of the primary server. This scheme puts a heavy burden on the recovery server. By contrast, in the primary backup scheme only one server is active at a time and the backup server becomes active only when the primary server fails. These two schemes suffer from either heavy recovery loads or low resource utilization. Our system uses multiple servers in parallel to provide video playbacks, as in the active-replication approach, but two recovery schemes are proposed to reduce failure-recovery overhead and prevent from low server utilization.

Figure 1 depicts the fundamental architecture of our multiple-server video-on-demand system. All the video servers are connected by a high-speed private network, called the

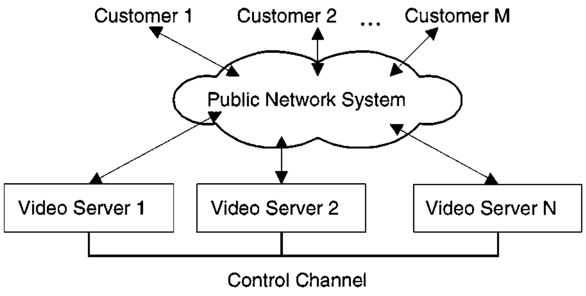


Figure 1. The architecture of the multiple-server video system.

control channel, which is used to exchange recovery and synchronization information among the video servers. Customers' commands are delivered to one of the video servers for a payback service. Note that in this paper the failure behaviour of the video server is assumed to be synchronous fail-silent [23, 24]. This behaviour constrains the failed server to stop providing video playbacks and to response to the other operational servers. The term "synchronous" means that the video servers can mutually detect the failure states within a certain time bound [3, 25].

Another challenge in implementing a video-on-demand system is providing continuous playback [14, 18, 20, 21]. A double-buffering mechanism is used to ensure video playback continuity [5, 22]. The concept behind is that: the video server allocates two buffers for each playback as an intermediate which separately manage video data. One buffer is being filled with video data from the storage while the other is sending data to customers. Both the buffers alternate their roles in turn until the playback is completed. Figure 2(a) illustrates the *video data access hierarchy* in a typical VOD environment. A *session* is defined as the channel with enough bandwidth for reading video data from disk storage to the system buffer. The bandwidth required to ensure playback continuity depends on the video format. For example, playing an MPEG-1 video requires the bandwidth of at least 1.5 Mb/s [8, 10].

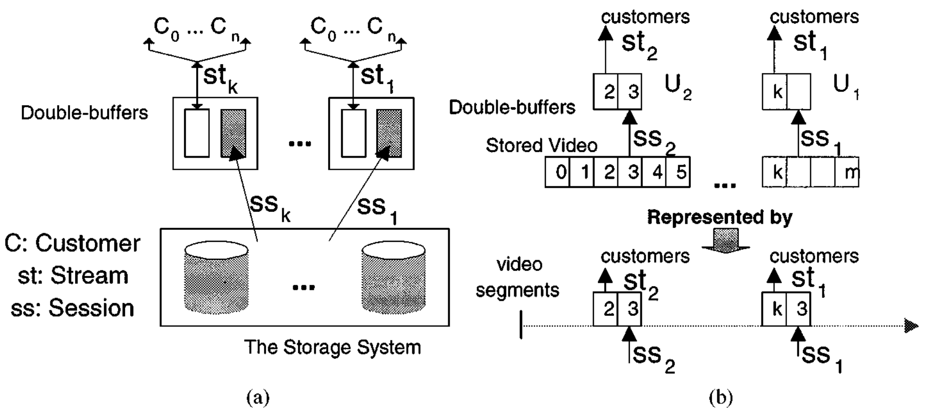


Figure 2. (a) Video data access hierarchy and (b) the simplified representation.

A *stream* is referred to as the channel for transferring video data to customers. A stream can support multiple customers by using a multicasting mechanism [1]. Resources, including a session, a stream and a set of double buffers, are defined as a *service unit*, denoted by U, in this paper. The power of a video server is determined by how many service units it can support simultaneously. A video playback is represented by the symbol $\mapsto$, as shown in figure 2(b). The vertical bar indicates the start point of a video, the arrow head indicates the end, and the length of the symbol the video length. We logically divide a video file in storage into continuous segments equal to the size of the intermediate buffer, video data thus can be transferred segment by segment in the access hierarchy.

This paper is organized as follows. In Section 2, we propose two distributed playback-recovery schemes, which guarantee minimal resources required for recovery since existing service units in the survival servers are used to deliver the interrupted playbacks originally provided by the faulty server. In Section 3, a distributed order-decision algorithm is proposed for on-line construction of order information for all distributed service units. Thus, when a server fails, this order information helps survival servers choose the service unit nearest to the faulty one for recovery. Section 4 presents three recovery policies that coordinate survival servers to consistently perform recovery process. In Section 5, simulation results show that our proposed schemes are feasible for enhancing fault-tolerant capability in video systems. Section 6 addresses our interesting research issues in the near future, and Section 7 concludes this paper. Step-by-step examination of the distributed order-decision algorithm and proofs of its correctness are presented in the appendix.

2. Distributed playback-recovery schemes

The simplest approach to the recovering of an interrupted playback in a faulty server is to allocate a new service unit in a survival server to continue providing playback instead. We refer to this method as an *allocation-based playback-recovery scheme*. Note that this scheme, while simple, entails great overhead because it demands additional service units from the survival servers. In this section, we proposed two recovery schemes to reduce such overhead, we first clarify some useful terminology before presenting these schemes. The service unit in a faulty server is called *faulty service unit*, and the service unit used for recovery in a survival server is called *recovery service unit*. A service unit contains two segment buffers. The least segment sequence number of a service unit, say U, is denoted by $S(U)$. For example, $S(U_2)$ as shown in figure 2(b) is 2. In addition, we also define a *leading relationship* for any two service units: given two service unit U_i and U_j, U_j leads U_i iff $S(U_i) < S(U_j)$, no matter they are located at same server or not.

2.1. *Forward playback-recovery scheme*

The idea behind the forward playback-recovery scheme is to recover from a faulty service unit by using an existing service unit that lags behind the faulty one. In this scheme, a survival server, having a running service unit which is playing the same video as one of the faulty service units, is responsible for recovering from this faulty one. That is, this server uses the running service unit as a recovery service unit. In addition, the forward

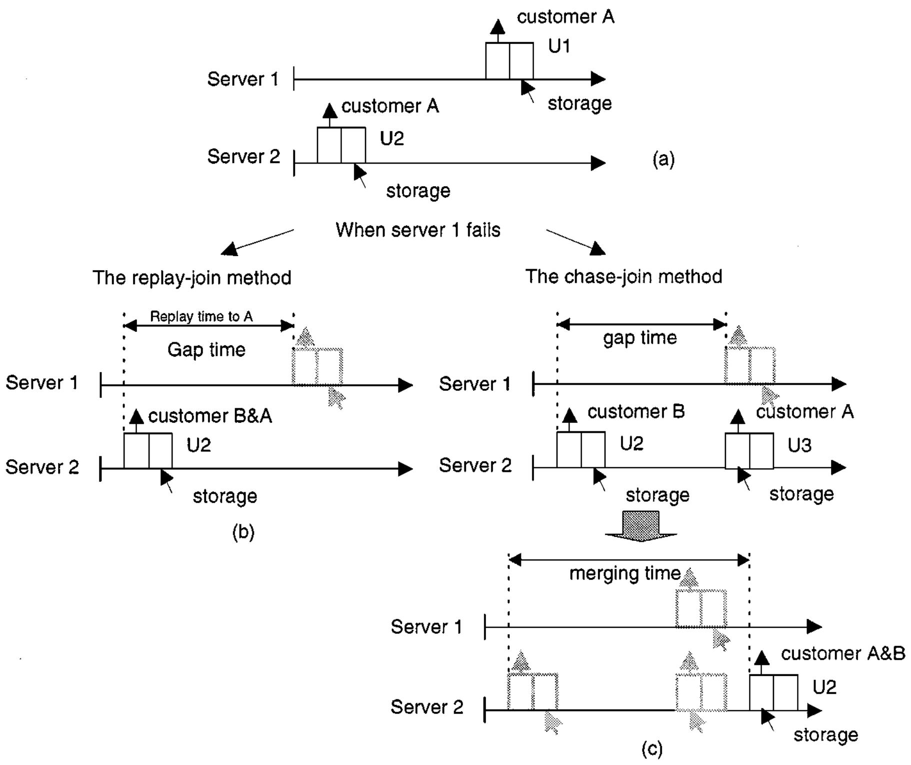


Figure 3. The forward playback-recovery scheme.

playback-recovery scheme requires the faulty service unit having a leading relationship to the recovery service unit. This scheme provides two methods to reduce the resources for recovery: one is the *replay-join method*, which directly delivers video data in the recovery service unit to customers originally served by the faulty service unit, the other one is the *chase-join method*, which allocates a temporary service unit in the survival server. This temporary service unit begins delivering data from the same segment as the faulty service unit, furthermore, its progress speed is adjusted to be reasonably slower so it can be merged with the recovery service unit later. Merging allows the temporary service unit to be released and thus saves a service unit. It is obvious that the replay-join method gives customers a repeated viewing because the faulty service unit leads the recovery service unit. We use an example to illustrate these methods. Consider the example shown in figure 3(a), where customers A and B are initially served by service units U_1 and U_2 in Server 1 and Server 2, respectively. Assume Server 1 fails. The replay-join method resumes customer A's playback by delivering video data from service unit U_2 as depicted in figure 3(b). That is, customers A and B share a common service unit. Note that customer A can continue watching the video, but will get a replay of $S(U_1) - S(U_2)$ segments. The temporal difference between $S(U_1)$ and $S(U_2)$ is defined as the *gap time*. That is, the gap time determines the length of video replay on the customers. The chase-join method allocates a temporary service unit U_3 to resume customer A's playback. In this example, customer A continues watching the video playback starting with the segment $S(U_1)$ from U_3 without any repeat. Furthermore, the progress speed of U_3 is adjusted to a slower rate that still gives acceptable visual perception. As U_2 catches up to U_3, these two service units are merged and U_3 is released. This method is transparent to customers but requires a temporary service unit for a period, this period is defined as the *merging time*, as shown in figure 3(c).

2.2. Backward playback-recovery scheme

In contrast with the forward playback-recovery scheme, the backward playback-recovery scheme uses an existing service unit which leads the faulty service unit as the recovery service unit (see figure 4(a)). Two join implementations can also be used with this scheme, however, both of them interfere with the progress of the foreground playback. In the replay-join method, existing service U_2 is used to recover from the faulty playback by delivering the video data from segment $S(U_1)$ (figure 4(b)). This forces customer B to watch a repeat from segment $S(U_1)$. The chase-join method also allocates a temporary service unit U_3 and delivers video from segment $S(U_1)$, as shown in figure 4(c). In this scheme, U_3's playback cannot be sped up due to the pre-determined bandwidth limitation of a service unit (although certain techniques can be used to accelerate the playbacks without requiring extra disk bandwidth by skipping the frames [2]. However, in this paper we assume that acceleration of a playback needs extra disk bandwidth. Thus, in order to merge the U_2 and U_3 playbacks, Server 2 slows down U_2's playback speed, interfering with the foreground playback quality.

Occasionally, the gap time between a recovery service unit and a faulty one will be so large that customers may have to watch excessively long replay periods. A time bound, called the *recovery time bound*, is defined to limit this problem. When it is used, only faulty service units, having a gap time shorter than the recovery time bound, will be recovered using the forward or backward playback-recovery scheme. If the time bound is exceeded, the allocation-based playback-recovery scheme is used instead, even though it requires extra service units. In Section 5, a simulation is used to show the recovery characteristics of these three schemes under a given recovery time bound.

The replay-join method does not need an extra service unit to perform recovery while the chase-join method does need a temporary one, but releases it later. Although these two recovery schemes mitigate recovery overhead by reducing the resources required, both methods also impose a period of recovery time (interval of replay or merging as shown in figures 3 and 4). However, it is a tradeoff between recovery time and recovery overhead for customers and service providers. From the customer viewpoint, the recovery time should be as short as possible, and this can be achieved by adjusting the playback speed of either the recovery service unit or the temporary service unit in advance. Golubchik et al. have provided adjustment techniques for merging two service units into one using adaptive piggybacking methods [13]. However, from the service provider's standpoint, it is desirable to recover as many faulty playbacks as possible. The ability to recover more playbacks can be enhanced by cutting down the number of service units required for recovery. This is an essential issue of the recovery schemes proposed above. Therefore, in this paper,

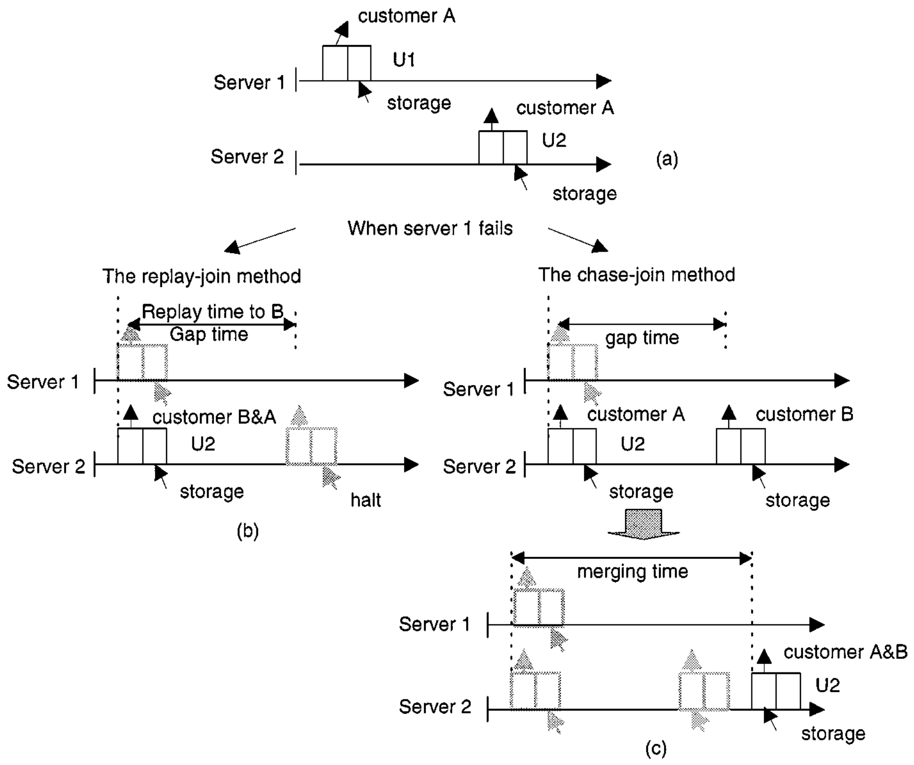


Figure 4. The backward playback-recovery scheme.

we focus primarily on how to conserve service units during recovery. Our approach is to shorten the recovery time by selecting a service unit nearest to a faulty service unit as the recovery service unit, and through the merging technique. However, the problem arising in a distributed environment is that these video servers lack a common clock mechanism to determine the distance and order information between service units across different servers. Therefore, in the next section, we will present a distributed algorithm that can order all the distributed service units according to the gap time of service units.

3. Distributed order-decision algorithm

In practice, the customers' requests arrive randomly at each video server, as shown in figure 1. Each server then allocates a service unit to provide video playback. Information concerning service units' order with respect to other servers are not maintained. As a result, when a server fails, it is difficult to effectively apply the above recovery schemes because we can not determine which running service unit is nearest to the faulty service unit. To solve this problem, we developed an algorithm called the *distributed order-decision algorithm* (DODA) to on-line construct the order relationship among the allocated service units across the distributed video servers.

3.1. Preliminaries and assumptions

This algorithm is accomplished by exchanging a set of messages, including the *setup* message, the *modification* message and the *termination* message, via the control channel. The maximal and minimal transmission time of the control channel are denoted by T_{maxDelay} and T_{minDelay}, which determine the precision of the gap and order information derived. The setup message informs the servers the allocation of a new service unit, the modification message modifies the order relationship constructed by the previous setup and modification messages when the order needs a change. The termination message informs the servers a playback is completed. A server need broadcast a setup message when it allocates a service unit to serve in response to a request. The setup message is denoted by sm_i^k and the broadcast time instant of this message by TB_i^k, where k stands for server k and i for the sequence number of this service unit. Each service unit is assigned a *sequence number* for identification. The server that broadcasts the setup message is referred to as the *sender* and the other servers as *receivers*. Using the arrival timeline of a setup message, as shown in figure 5, we can determine the order relationship between the service units of the sender and receiver. The timeline of a receiver can be partitioned into two types of intervals, the *distinguishable time interval* and the *indistinguishable time interval*. The indistinguishable time interval, TI_j, is an interval from the creation time of service unit j plus T_{minDelay} to the creation time plus T_{maxDelay} (the creation time is the time that service unit i is allocated and the time instant broadcasting a setup message for this service unit). The distinguishable time interval, TD_{j-1}, is an interval from the creation time of service unit $j-1$ plus T_{maxDelay} to the next service unit j's creation time plus T_{minDelay}. In figure 5(a), the white circles represent the time instant at which a setup message is broadcasted and *TA* represents the time instant at which a setup message arrives at a server. The horizontal axes in the figure represent the time lines for each server. Accordingly, we can derive three order relationships between any two service units: (1) *Sender Leading Receiver* (SLR), (2) *Receiver Leading Sender* (RLS) and (3) *Concurrent*. If a receiver receives a setup message, say sm_i, within the distinguishable interval TD_{j-1}, we can infer the service unit U_i is leading the service unit U_j if the latest created service unit in the receiver is U_j, because the minimal transmission time of sm_i is larger than T_{minDelay}. Thus, these two service units have an SLR relationship and the property $S(U_i) > S(U_j)$ holds, as shown in figure 5(a). If a setup message, say sm_i arrives

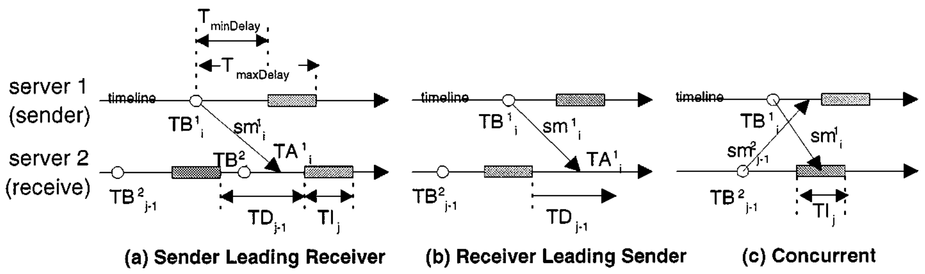


Figure 5. Relationships between sender and receiver.

at a receiver within the distinguishable interval say TD_{j-1}, and the most recent created service unit in the receiver is U_{j-1}, we can infer that the receiver's latest service unit U_{j-1} leads U_i , because the maximal transmission time of sm_i won't be greater than T_{maxDelay}. These two service units have an RLS relationship and the property $S(U_{j-1}) > S(U_i)$ holds, as shown in figure 5(b). If a setup message arrives at a receiver within the indistinguishable interval, say TI_{j-1}, the receiver cannot determine the order relationship for U_i and U_{j-1}. In this case, the receiver considers U_i and U_{j-1} to be concurrent. Figure 5(c) shows the concurrent relationship. However, the sender (Server 1) may figure out a SLR relationship if sm_{j-1} arrives server 1 within TD_{i-1}. Therefore in order to eliminate the inconsistent recognition, extra modification message is needed to negotiate. In Section 3.2, we will explain the modification process in detail.

A message consists of four fields: ⟨*server ID*⟩, ⟨*type*⟩, ⟨*vector*⟩ and ⟨*checkpoint data*⟩. The ⟨server ID⟩ field indicates which server sent the message. The ⟨type⟩ field identifies the message as a setup, modification or termination type. The modification message is further classified as either an SLR or RLS modification. Only a setup message has the ⟨checkpoint⟩ field, which contains the stream and customer information of the service unit. The ⟨vector⟩ field has three n-entry vectors: the *order vector* (V), the *forward gap-time vector* (F) and the *backward gap-time vector* (B), where n is the number of servers. The order vector records information concerning about the order relationship known to the sender, in which each entry represents the highest known service unit's sequence number for each other individual server. For example, an order vector $V = [e_x, e_y, e_z]_2$, sent by Server 2, informs the receivers that server 2's $U^2_{e_y}$ is ordered behind $U^1_{e_x}$ in Server 1 and $U^3_{e_z}$ in Server 3. This also implies that U^1_0 to $U^1_{e_x}$ in Server 1 lead $U^2_{e_y}$ and so do in Server 3. The forward and backward gap-time vectors record the approximate gap times between current service units and those ahead of it. The difference is that the forward gap-time vector is approximated by the preceding service unit and the backward gap-time vector by succeeding service unit. For example, a backward gap-time vector $B = [g^b_x, g^b_y, g^b_z]$ in conjunction with the above V represents that the gap time between $U^2_{e_y}$ and $U^1_{e_x}$ is approximated as g^b_x by Server 1, that between $U^2_{e_y}$ and $U^3_{e_z}$ as g^b_z by Server 3. Note that g^b_y is the gap time between $U^2_{e_y}$ and $U^2_{e_y-1}$. A forward gap-time vector $F = [g^f_x, g^f_y, g^f_z]$ in conjunction with above V represents that the gap time between $U^2_{e_y}$ and $U^1_{e_x}$ is approximated as g^f_x and that between $U^2_{e_y}$ and $U^3_{e_z}$ as g^f_z by Server 2. This algorithm is quite complex, in the appendix we explain in detail about how to generate these vectors. In the following section, we briefly describe the key ideas behind our algorithm.

3.2. Algorithm

In DODA, two phases of negotiation are needed before a complete order information is constructed. The first phase, *Absolute Order Generation*, yields an absolute order relationship by exchanging setup and modification messages between servers. However, this absolute order does not suffice to construct a unique ordered sequence for the service units across video servers because of the concurrent relationship. This insufficiency is thus compensated for by the another phase, *Relative Order Generation*.

Absolute order generation. This phase generates the leading relationship for all the distributed service units. Each server broadcasts a setup message to other servers only when a new service unit is allocated in response to a playback request. The broadcasting server constructs a setup message that contains an order vector recording the highest sequence numbers of service units of all other servers, and forward and backward gap-time vectors which records the approximate gap time information. While receiving a setup message, the receiver checks which interval the message arrived within. According to the arrival interval, the receiver can determine the order between the service unit represented by the setup message and its most recent service unit. If the setup message is received within a distinguishable time interval, SLR or RLS relationship can be obtained, otherwise, concurrent relationship is recorded. For example, in the SLR case as shown in figure 5(a), the order vector of U_i^1 in Server 1 enclosed in sm_i^1, may contains two possible order information, either $V_i^1 = [i, j-1]_1$ or $V_i^1 = [i, j-2]_1$. The vector reflects the order information inferred by server i. If the order vector is $V_i^1 = [i, j-1]_1$, it represents that server 1 only knows that U_{j-1}^2 led U_i^1 when U_i^1 was created. The result is consistent with server 2's information, thus need not send any modification message. If the order vector is $V_i^1 = [i, j-2]_1$, it means that Server 1 had not yet received a setup message of creating U_{j-1}^2, or considers the relationship between U_{j-2}^1 and U_i^2 to be concurrent. In order to rectify this incorrect order relationship, Server 2 should broadcast a modification message that contains a vector $V_i^1 = [i, j-1]_1$ to inform other servers that U_{j-1}^2 led U_i^1. Furthermore, server 2 must also also construct a modification message for U_j^2 because sm_j^2 sent incorrect order information $V_j^2 = [i-1, j]_2$. Thus, while receiving sm_i^1 within TD_{j-1}^2, Server 2 broadcasts a modification message containing an order vector $V_j^2 = [i, j]_2$ to inform other servers that U_i^1 should lead U_j^2.

Relative order generation. If all setup messages are received within distinguishable time intervals, the order relationships among service units can be determined easily and correctly. Unfortunately, ordering is sometimes unclear due to the concurrent relationship. This makes the relative order relationship difficult to be determined when setup messages are received within indistinguishable time intervals. For example, given two setup messages sm_i^1 and sm_{j-1}^2 containing order vectors, $V_i^1 = [i, j-2]_1$ and $V_{j-1}^2 = [i-1, j-1]_2$, respectively, and both of them are received within indistinguishable time intervals, TI_{j-1}^2 and TI_i^1, as shown in figure 5(c). In this case, no adequate information is available to determine which one goes first. A simple way to resolve this concurrent problem is to assume that the server with the smaller server ID number has the precedence. This assumption is referred to as the *small server identification first*. Therefore, U_i^1 would be considered to lead U_{j-1}^2. The detail is described in the *Order-Decision* procedure given in Appendix A.1.

3.3. Computation of the gap time

The precise gap time between any two successive service units can be computed by subtracting their creation time (allocation time) instants, however, in a distributed environment, a server is unable to determine the exact creation time instant of another server's service unit. A safe way to conserve the precision is to approximate the gap time by subtracting

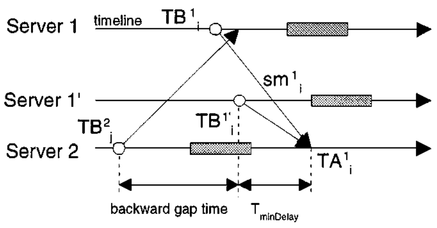


Figure 6. Computation of the gap time.

a T_{minDelay} from its arrival time. As an example in figure 6, Server 2 approximates U^1_i's creation time by subtracting a T_{minDelay} from TA^1_i. Thus, the backward gap time between U^2_j and U^1_i is equal to $(TA^1_i - T_{\text{minDelay}}) - TB^2_j$. The time line of server $1'$ is an illusion of Server 1 considered by Server 2. If Server 1 fails, Server 2 can estimate the length of replay or merge time from the backward gap time to determine whether U^2_j is the best candidate to recover U^1_i. The forward gap time is calculated by Server 1 using $TB^2_j - (TA^1_i - T_{\text{minDelay}})$. The subtraction of a T_{minDelay} ensures that the calculated gap time will not be over-estimated though it does result in playback repetition for customers. These mechanisms are listed in the *Send* and *Receive* procedures of phase 1, and given in the appendix.

3.4. Order expression

Each server executes the distributed order decision algorithm individually and interacts with the other servers by exchanging messages. A newly allocated service unit causes the server to broadcast a setup message and executes this algorithm. After two phases described above have been completed, the order and gap time information is obtained by interpreting these enclosed vectors, and can be represented as:

$$U_1 \underset{g^f_2}{\overset{g^b_1}{\underset{\longleftarrow f}{\overset{b\longrightarrow}{}}}} U_2, \ldots, U_{n-1} \underset{g^f_n}{\overset{g^b_{n-1}}{\underset{\longleftarrow f}{\overset{b\longrightarrow}{}}}} U_n$$

which is referred to as *recovery information*, where g^b and g^f are the gap time between the two successive service units. The symbols $b \rightarrow$ and $\leftarrow f$ stand for the gap time, g^b and g^f, respectively, as calculated by succeeding and preceding service units. The recovery information represents that U_1 leads U_2, U_2 leads U_3 and so on. Through DODA coordination, all video servers share a common view of the recovery information about all distributed service units. Thus, when a server suffers a failure, the other survival servers can use this consistent information to perform recovery.

4. Server failure treatment

In Section 2, we proposed two playback-recovery schemes, the forward playback-recovery scheme and the backward playback-recovery scheme, to handle recovery process. These two recovery schemes greatly reduce the recovery overhead through the replay-join method or the chase-join method, however, there is still playback repetition or resource requirement (allocation of temporary service units). Thus, proper selection of a suitable recovery scheme as well as a join method for the faulty service unit will make the length of the playback repetition and the holding time of the temporary service unit shorter.

4.1. *Playback recovery*

Recovery information is useful in helping the forward playback-recovery scheme and the backward playback-recovery scheme choose a closest recovery service unit. For example, given any two successive service units with a recovery information $U_j \begin{smallmatrix} g_j^b \\ b\longrightarrow \\ \longleftarrow f \\ g_i^f \end{smallmatrix} U_i$. According to this recovery information, the video server could use U_i to recover U_j by using the forward playback-recovery scheme when U_j becomes a faulty service unit, and the cost is g_i^f. That is, with the replay-join method, a replay of g_i^f is for the customers on U_j; and with the chase-join method, a temporary service unit would retrieve video data from the time instant of $S(U_i) - g_i^f$, and the merging time is also proportional to the gap time g_i^f. Similarly, If the U_i becomes a faulty service unit, U_j can be the recovery service unit by the backward playback-recovery scheme and the recovery cost is g_j^b. As a result, we can obtain overhead criteria of each recovery scheme from the recovery information and are thus able to select a good recovery scheme for each faulty service unit. In the next section, we state three policies for synchronizing the scheme selection among video servers.

4.2. *Recovery scheme selection criteria*

In order to perform recovery correctly, survival servers must have a policy to coordinate the selection of the recovery service units for those faulty service units, otherwise some faulty service units would be recovered by more than one service units or none. Below, we present three recovery policies to guide the servers in response to a server failure.

Forward policy. The forward policy is highly intuitive. Before any failure occurs, all video servers negotiate to apply the forward playback-recovery scheme in response to a server failure. Therefore, when the servers detect a failure in some video server, they inspect their recovery information to see which service units are ordered behind the faulty ones, furthermore, the servers having the closest service units to the faulty ones are responsible for the recovery of those faulty service units by applying the forward playback-recovery scheme. Because all servers have the identical recovery information, every faulty service unit is guaranteed to be recovered exactly once by its closest service unit. Note that if the last service unit in the recovery information is a faulty one, the server having a service unit preceding the faulty one would use the backward playback-recovery scheme instead in such a case.

Backward policy. In this policy, all the video servers negotiate to use the backward playback-recovery scheme instead of the forward playback-recovery scheme when a server failure occurs, and note that if the first service unit in the recovery information is faulty, a following service unit uses the forward playback-recovery scheme instead.

Hybrid policy. The hybrid policy combines the forward and backward playback-recovery schemes by selecting the service unit with the smallest gap time as the recovery service unit. Thus, a faulty service unit would be recovered by using the forward playback-recovery scheme or the backward playback-recovery scheme. Because the recovery information are identical for all the video servers, this policy won't result in conflicts or starvation in recovering for the faulty service units.

4.3. Fault-tolerant resiliency

The distributed order-decision algorithm can tolerate server failures because order and gap time information is exchanged in a one-to-one manner across servers. Thus, a server failure does not affect the correctness of the algorithm, nor do the survival servers have to restart the algorithm. In addition, the order and gap vectors are still correct when the service units of the faulty server are removed from the recovery information. As an example shown in Appendix A.3, given a recovery information as follows:

$$U_0^2 \mathrel{\substack{b\xrightarrow{30} \\ \xleftarrow[19]{}f}} U_2^3 \mathrel{\substack{b\xrightarrow{13} \\ \xleftarrow[0]{}f}} U_0^1 \mathrel{\substack{b\xrightarrow{0} \\ \xleftarrow[0]{}f}} U_1^2 \mathrel{\substack{b\xrightarrow{32} \\ \xleftarrow[18]{}f}} U_1^1 \mathrel{\substack{b\xrightarrow{18} \\ \xleftarrow[9]{}f}} U_1^3 \cdots$$

Assuming Server 2 fails, the service units in Server 2 U_0^2 and U_1^2 are removed and the recovery information is thus interpreted from the original order and gap vectors as:

$$U_0^3 \mathrel{\substack{b\xrightarrow{13} \\ \xleftarrow[0]{}f}} U_0^1 \mathrel{\substack{b\xrightarrow{23} \\ \xleftarrow[23]{}f}} U_1^1 \mathrel{\substack{b\xrightarrow{18} \\ \xleftarrow[9]{}f}} U_1^3 \cdots$$

Moreover, the survival servers will still tolerate server failures until only one server is left in operation. With the distributed order-decision algorithm, we can tolerate $n - 1$ server sequential failures in a n-server VOD system.

5. Simulation

To compare the various recovery policies, two simulation studies were conducted on an Intel Pentium-based machine with 64 MB of RAM. The simulation architecture is illustrated in figure 1. Because the customer can request a playback on any one of the video servers, we simulated such behavior by generating a Poisson arrival process characterized by the mean interarrival intervals for each server. In the simulation process, the system used ten video objects as materials for simulating a video-on-demand system. For each arrived request, we used a predefined object's access frequency to determine which video object the request asked. The access frequencies to various objects were characterized by a *Zipf* distribution with the parameter 0.1386. (In a *Zipf* distribution, if objects are sorted according to access frequency, then access frequency for the ith object is given by $f_i = \frac{c}{i^{1-\theta}}$, where θ is the parameter for the distribution and c is the normalization constant [4, 17, 26]. The assignment of 0.1386 to θ recognizes that 80% of the requests will ask for 20% of the objects, and the remaining 20% of the requests will ask for the other 80% of the objects, a phenomena called the 80-20 rule. Using this rule makes a simulated video system seem more realistic by allowing for the different degree of popularity between popular videos and cold videos.) After determining a target video, a request waited in a queue until a free service unit was available. The server then allocated a service unit to provide playback. Each video server had the ability to provide 100 service units simultaneously. The maximal and minimal transmission times for the control channel were assumed to 500 and 50 ms, respectively. The video lengths were 1.5 h.

In the first simulation, a video system with only one video was simulated, that is, all the requests asked for the same video. The average gap time versus the requests' interarrival time was considered. We used G_i to represent the *average server gap time* for recovering from server i's faulty service units under server i failure. This was calculated by dividing the summation of all the used gap time by the number of faulty service units. The *average system gap time* $\bar{G}_{\text{system}}$ was furthermore calculated by the equation:

$$\bar{G}_{\text{system}} = \frac{\sum_{j=1}^{N} G_i}{N}, \text{ where } N \text{ was the total number of servers.}$$

By assigning each Poisson process a different interarrival interval, we simulated this video- on-demand system for 100 h, and then calculated the respective average system gap times for the forward policy, the backward policy and hybrid policy. Figure 7 shows the average system gap time for various request interarrival intervals in video systems with two, four, six and eight servers, respectively. The statistical diagrams show that the hybrid policy kept the average system gap time shorter than the other policies. This means that using the hybrid policy will result in lower recovery overhead or shorter video replays. Moreover, we found that the more the servers involved, the shorter the average system gap time was. We can thus conclude that a system can provide more playbacks simultaneously, and quicker playback recoveries in the presence of failures if the overall number of video servers is increased.

In the second simulation, the recovery time bound on the recoverable service unit was considered. Under a given recovery time bound, we evaluated how many faulty service units could be recovered using the forward or backward playback-recovery schemes. If the gap time for recovering a faulty service unit was larger than the given recovery time bound, an allocation-based recovery scheme was applied instead, which required extra service units until the playback was completed. In this simulation, we increased the number of videos to 10 and fixed the interarrival intervals for each Poisson process at 10 min. This simulation was continued for 10,000 h, and figure 8 shows the evaluation results. We found that the hybrid policy allowed more faulty service units to be recovered by mixing the forward and backward recovery schemes, and the number of recoverable service units increased as the recovery time bound was increased. From those results we found that a lot of service units

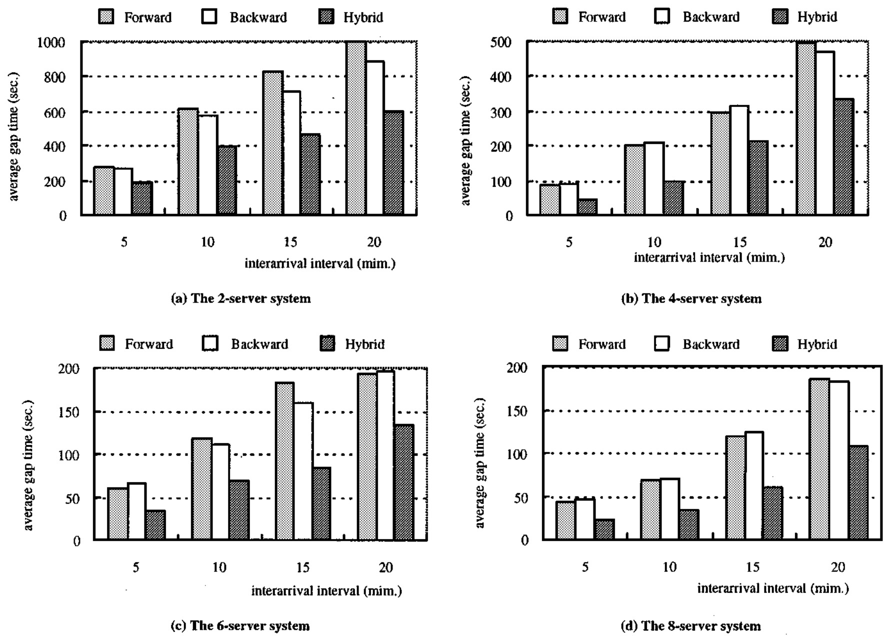


Figure 7. The average system gap time vs. the requests interarrival interval.

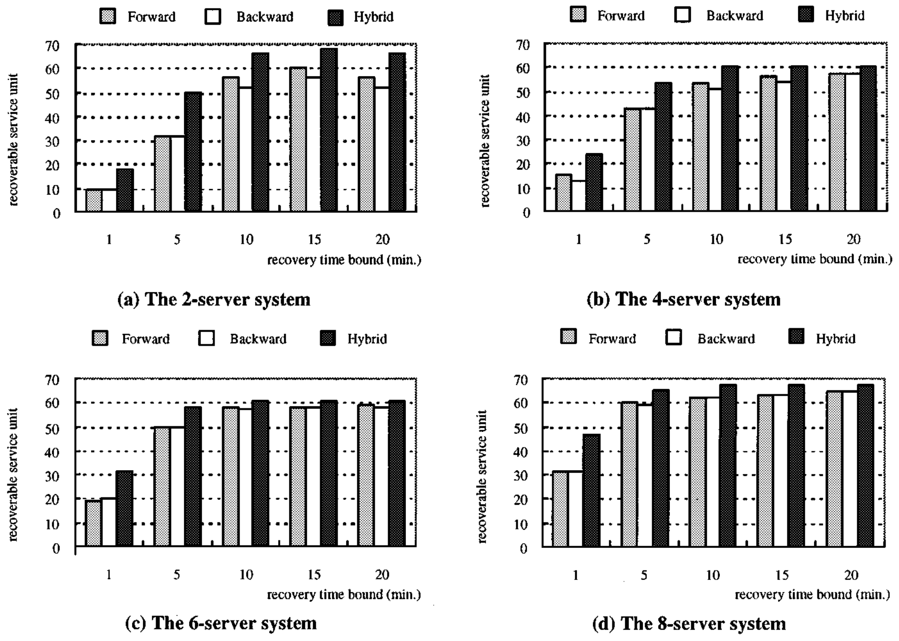


Figure 8. The recoverable service unit vs. the recovery time bound.

were saved by preventing invocation of the allocation-based recovery scheme. And, the greater the number of servers, the greater the number of recoverable service units was, even with shorter recovery time bounds.

6. Future work

This paper proposed a fault-recovery methodology for a distributed video-on-demand system under the assumption that a common-clock synchronization mechanism among the video servers was lacking. Thus, an algorithm was needed to determine the order relationship across the faulty and recovery service units. Another research issue is to provide a fault-tolerant design that does use a global synchronization mechanism. This mechanism can be realized in two ways: (1) each video server is equipped with a permanent highly accurate clock synchronized with some single official and legal time; e.g., UTC, as received from standardization organizations via terrestrial radio stations, or the *Global Positioning System*, or (2) all requests are first delivered to a central server which then dispatches these requests to various video servers. Through the centralized control, service-unit order can be determined in advance. The design methodology for such a system will vary from the methods discussed in this paper. Furthermore, how to balance the recovery loads among the survival servers after a server failure is also an important topic. Thus, in the near future we will focus on balancing recovery loads and providing a fault-tolerant design that includes a global synchronization mechanism.

7. Conclusion

In this paper, we proposed forward and backward playback-recovery schemes, to cope with server failures in a distributed video-on-demand system. These schemes use existing resources to perform recovery process and thus significantly reduce recovery overhead on survival servers. In addition, we developed a distributed algorithm, called the distributed order-decision algorithm that on-line constructs the order and gap information for the distributed service units. This recovery information is useful in helping our proposed schemes find the nearest recovery service units. We also proposed three recovery policies to guide recovery servers in applying suitable recovery schemes using this recovery information. Finally, two simulations were conducted, and the results showed that the recovery schemes are feasible and effective for use in a realistic video-on-demand system.

Appendix

A.1. The distributed order-decision algorithm (DODA)

There are four primary data structures maintained by each video server: the *sequence variables*, the *state transition list* (STL), the *order decision list* (ODL) and the *total order list* (TOL). In addition, a server, say k, maintains n sequence variables S_i^k, where $1 \leq i \leq n$; S_k^k records the sequence number of the latest allocated service unit in server k. The other variables record the largest sequence numbers of the service units already received from the other servers.

Absolute order generation. The first part of DODA consists of three procedures: *Init* procedure, *Send* procedure and *Receive* procedure. The *Init* procedure is executed once to initiate data structures. The *Send* procedure is invoked to broadcast a setup message when a server allocates a new service unit. The purpose of the *Send* procedure is to construct the order vector which contains the information about which service units are running behind the newly allocated service unit. The forward and backward gap time vectors are also computed here according to the method in Section 3.3. The *Receive* procedure reacts to the received messages according to its type and arrival time.

```
Procedure Init (server $k$)
begin
    1. $STL$, $ODL$ and $TOL$ are empty initially.
    2. $S_i^k$ is set to $-1$, where $i$ from 1 to $n$ for server $k$.
end
```

In the *Send* procedure, the order vector is constructed by filling each entry with the largest sequence number of the received service units from other servers individually, all of them lead the current newly allocated one. The largest sequence number of the received service units from server i is recorded as S_i. This procedure also constructs forward and backward gap time vectors. The parameter server k represents the caller of this procedure. Server k and the broadcasting instant t of the setup message (TB is equal to TA for the broadcasting server).

```
Procedure Send ($k$)
begin
    1. Generate an order vector $V = [e_1, e_2, \ldots, e_n]_k$, satisfies
        a. $e_k = S_k^k + 1$ and,
        b. $e_i = S_i^k$, where $i \neq k$ and $1 \leq i \leq n$.
    2. Generate a forward gap time vector $F = [f_1, f_2, \ldots, f_n]$, satisfies:
        a. if ($S_k^k == -1$)
                $f_k = -1$;
            else
                $f_k = t - TA(sm^k_{S_k^k})$;
        b. if ($e_i == -1$)
                $f_i = -1$;
            else
                $f_i = t - (TA(sm^i_{e_i}) - T_{\text{minDelay}})$, where $i \neq k$ and $1 \leq i \leq n$;
    3. Generate a backward gap time vector $B = [b_1, b_2, \ldots, b_n]$, satisfies:
        a. if ($S_k^k == -1$)
                $b_k = -1$;
            else
                $b_k = t - TA(sm^k_{S^k_k})$
        b. $b_i = -1$, where $i \neq k$ and $1 \leq i \leq n$.
    4. Broadcast a setup message $(k$, Setup, $V$, $F$, $B)$.
    5. Insert $(TA(t)$, $V$, $F$ $B)$ to $STL$, where $t$ is the time instant of step 4.
    6. $S^k_k = S^k_k + 1$.
end
```

This *Receive* procedure checks whether the message is received within the distinguishable time interval or the indistinguishable time interval, and whether its type is *setup*, *modification* or *termination*. Different responses must be made depending on the type and the arrival time. The *Receive* procedure shown below describes the actions for server h which receives a message m at time t_a.

```
Procedure Receive $(h, m, t_a)$
$m$ := received message, $(k$, $type$, $V_k$, $F$, $B)$;
$k$ := sender;
$type$ := message type;
$V$ := order vector, where $V = [e_1, e_2, \ldots, e_k, \ldots, e_n]_k$.
$F$ := forward gap time vector, where $F = [f_1, f_2, \ldots, f_k, \ldots, f_n]$.
$B$ := backward gap time vector, where $B = [b_1, b_2, \ldots, b_k, \ldots, b_n]$.
$V_t$, $F_t$, and $B_t$ refer to the order, forward gap time and backward gap time vectors in a
    node of $STL^h$.
begin
    switch ($type$)
    case: (Setup)
        1. $S^h_k = e_k$.
        2. if $(S^h_h == -1)$    /* server h has not allocated any service unit yet */
            a. Insert $(TA(t_a)$, $V$, $F$, $B)$ to $STL^h$.
            b. exit.
        3. if $(t_a < TA(sm^h_{S^h_h}) + T_{\mathrm{maxDelay}})$    /* the case of RLS, the message arrives within */
                a. if $(e_h \neq S^h_h)$    /* the order relationship is not correct, so broadcast a RLS
                                              modification message to revise it */
                    a.1 $e_h = S^h_h$.
                    a.2 $b_h = (t_a - T_{\mathrm{minDelay}}) - TA(sm^h_{S^h_h})$.
                    a.3 Broadcast $(h, Modif._{RLS}, V, F, B)$.
                else if $(e_h \neq -1)$    /* the order reported is correct, now calculate the back-
                                              ward gap time vector */
                    a.4 $b_h = (t_a - T_{\mathrm{minDelay}}) - TA(sm^h_{S^h_h})$.
                    a.5 Broadcast $(h$, $Modification$, $V$, $F$, $B)$.
                b. Append $(TA(t_a)$, $V$, $F$, $B)$ to $STL^h$.
                c. exit.
    4. if ($t_a < TA(sm^h_{S^h_h}) + T_{\text{minDelay}}$)
        /* the case of SLR, the message arrives within $TD^h_{S^h_h-1}$ */.
            the server h should reflect that the service unit $e_k$ runs ahead of service unit
              $S^h_h$ due to the SLR relation */
            a. Find the vectors $V_t$, $F_t$, $B_t$ from the node which is for $sm^h_{s^h_h}$ in $STL^h$
                a.1 The $h$th entry of $V_t$ is modified to $S^h_h$.
                a.2 The $h$th entry of $F_t$ is modified to $TA(sm^h_{S^h_h}) - (t_a - T_{\text{minDelay}})$.
                a.3 The $h$th entry of $B_t$ is modified to $-1$.
                a.4 Broadcast ($h$, $Modi._{SLR}$, $V_t$, $F_t$, $B_t$).
            /* calculate the backward time gap between the server k's service unit $e_k$
                and the server h's service unit $e_h$ */
            b. if ($e_h \neq -1$)
                b.1 $b_h = (t_a - T_{\text{minDelay}}) - TA(sm^h_{e_h})$.
                b.2 Broadcast ($h$, $Modification$, $V$, $F$, $B$).
            c. Append ($TA(t_a)$, $V$, $F$, $B$) to $STL^h$.
            d. exit.
    5. if ($TA(sm^h_{S^h_h}) + T_{\text{minDelay}} \leq T_a \leq TA(sm^h_{S^h_h}) + T_{\text{maxDelay}}$)
        /* the case of concurrent */
            a. if ($e_h \neq -1$)    /* only calculate the backward time gap */
                a.1 $b_h = (t_a - T_{\text{minDelay}}) - TA(sm^h_{e_h})$.
                a.2 Broadcast ($h$, $Modification$, $V$, $F$, $B$).
            b. Append ($TA(t_a)$, $V$, $F$, $B$) to $STL^h$.
            c. exit.
case: (Modification)
    1. Find the vectors $V_t$, $F_t$ and $B_t$ from the node which is for $sm^k_{e_k}$ in $STL^h$.
    2. if ($V == V_t$)
        /* if neither SLR or RLS case, only gap time is modified */
            a. Update the newest entries in $F$ and $B$ to $F_t$ and $B_t$ respectively.
            b. exit.
    3. if ($Modi._{SLR}$)
            a. if ($e_h \neq$ the $h$th entry of $V_t$)
                /* the case of SLR order and gap modification */
                a.1 $b_h$ is modified to $(TA(sm^k_{e_k}) - T_{\text{minDelay}}) - TA(sm^h_{e_h})$.
                a.2 Broadcast a modification message, ($h$, $Modi.$, $V$, $F$, $B$ ) servers.
            b. Update the newest entries in $V$, $F$ and $B$ to $V_t$, $F_t$ and $B_t$, respectively.
            c. exit.
    4. if ($Modi.RLS$)
            a. if ($k == h$ and $e_i \neq$ the $i$th entry of $V_t$, where $i \neq h$)
                /* for RLS order and gap modification */
                a.1 $f_i = MAX(0, TA(sm^k_{e_k}) - (TA(sm^i_{e_i}) - T_{\text{minDelay}}))$.
                a.2 Broadcast ($h$, $Modification$, $V$, $F$, $B$).
            b. Update the newest entries in $V$, $F$ and $B$ to $V_t$, $F_t$ and $B_t$, respectively.
            c. exit.
```

```
  case: (Termination.)
    1. Remove the corresponding vectors in $TOL^h$ to reflect the close of a playback.
    2. exit.
end
```

The content of a vector may be transient in STL, because further modification messages would revise the vector entries. The modification corrects the confusion which arises when one server recognizes a concurrent relationship but another recognizes an SLR or RLS relationship. The modification process continues for at most three times T_{maxDelay} in the algorithm. Formally, the content of vectors becomes stable after $3 \times T_{\text{maxDelay}}$ from its allocation time because the conditions of RLS and SLR require at most three times T_{maxDelay} to revise the origin vectors. Namely, the node, containing (TA, V, F, B), is kept in STL for at most three times T_{maxDelay}. Note that for the vector, which represents the service unit from other servers, is only kept in STL for $3 \times T_{\text{maxDelay}} - T_{\text{minDelay}}$. And then this node is moved to ODL.

Relative order generation. As described above, each node in STL is then moved to ODL by executing the *Push* procedure after staying in *STL* for $3 \times T_{\text{maxDelay}}$ or $3 \times T_{\text{maxDelay}} - T_{\text{minDelay}}$.

```
Procedure Push ($k$, node $(TA, V, F, B)$)
begin
  1. Append $(TA, V, F, B)$ to $ODL^k$.
  2. Set a timer on this node with one $T_{\text{maxDelay}}$ time quantum.
end
```

During the decision process all the vectors confused with each others must be considered together, or the order will be incorrect. The purpose of setting a T_{maxDelay} timer is to ensure that the condition will be met. In the later, we will prove that the vectors with a concurrent relationship will arrive within T_{maxDelay} after the first arrival in these setup messages with a concurrent relation. When each node's timer is expired, the *Order-Decision* procedure is executed and a domination group is formed. The *domination group* of a service unit e_z with an order vector, say $[e_x, e_y, e_z]_3$, is defined as the set of service units including U_0^1 to $U_{e_x}^1$, U_0^2 to $U_{e_y}^2$ and U_0^3 to $U_{e_z-1}^3$. If the members of a domination group for a given service unit are all in TOL, this service unit is called *pop-able* from ODL and is capable of being moved to TOL. The *Order-Decision* procedure is invoked by the node in which the timer is expired.

```
Procedure Order-Decision ($k$, node $(TA, V, F, B)$)
begin
  repeat
    1. Find all the pop-able service units in $ODL^k$.
    2. Sort these pop-able service units by the small server's identification first scheme.
```

```
        3. Extract the first pop-able service unit's node from $ODL^k$ and append it to the tail
           of $TOL^k$.
    until (($TA$, $V$, $F$, $B$) is the just appended node) or (there is no pop-able service unit).
end
```

A.2. Translation from TOL to recovery information

In the first phase of DODA, each node stays in STL for at most $3 \times T_{\text{maxDelay}}$. In the second phase, the node stays for only T_{maxDelay}. Thus we can conclude that for any newly allocated service unit, our algorithm takes no more than $4 \times T_{\text{maxDelay}}$ to resolve each service unit's order and gap time against to all others from its allocation. This recovery information to other service units can be interpreted from TOL. Each node in TOL represents a service unit. Thus, the order in TOL stands for the order of a service unit. For any successive nodes in TOL, assume that they have the formats of $Node_x = (TA_x, V_i, F_i, B_i)$ and $Node_y = (TA_y, V_j, F_j, B_j)$, where i and j denote which servers the vectors belong to. Thus if $Node_x$ is ordered ahead of $Node_y$ in TOL, it means that the service unit that the service unit $V_i[i]$ leads the service unit $V_j[j]$, where "[ ]" means the indexed entry of the vector, and the backward gap time is $B_i[j]$ and the forward gap time is $F_j[i]$. We can interpret the recovery information as follows:

$$U^i_{V_i[i]} \underset{F_j[j]}{\overset{B_j[j]}{\overset{b\longrightarrow}{\longleftarrow f}}} U^j_{V_j[j]}$$

A.3. Example

The example shown in figure 9 is used to illustrate DODA. Assume there are three video servers in our VOD system. Each server executes DODA and interacts with others by exchanging messages. The time line is divided into smaller time units. The maximal and minimal transmission time are 17 and 6 time units, respectively. We show the message flows according their time event sequences. Playback requests arrived at different servers and the servers allocate service units to serve these requests individually. A setup message is broadcasted to all servers for each newly allocated service unit. For example, server 2

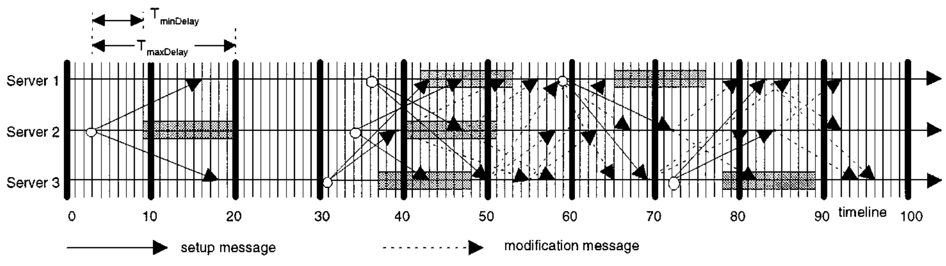


Figure 9. A 3-server video-on-demand system.

broadcasts sm_0^2 to server 1 and server 3 for U_0^2. Its arrival time at these servers are 16, 3 and 18, respectively.

Table 1 shows the broadcasted and received messages to each server, sorted by their occurring time. Rc. and Br. are the abbreviations of the terms “receive” and “broadcast”. Table 2 shows server 1’s transitions about the sequence variables, the state transition list, the order decision list and the total order list according to the message flows. As described in the first phase, each node stays in STL for $3 \times T_{\text{maxDelay}}$ or $3 \times T_{\text{maxDelay}} - T_{\text{minDelay}}$. We calculate each node’s leaving time from STL to ODL. For example, at the 16th time unit, the node of $TA(16)$, $V[\text{Nil}, 0, \text{Nil}]_2 F[-1, -1, -1]$ $B[-1, -1, -1]$ is appended to STL in Server 1, its leaving time from STL to ODL is calculated as the 61th time unit $(16 + 17 \times 3 - 6 = 61)$.

Through DODA, each server will progressively obtain the same order and gap time image of all the distributed service units. Our algorithm assures all the servers will obtain the new order and gap time of a service unit after four times T_{maxDealy} from its allocation. This example generates the following recovery information for those distributed service units at the 100th time unit.

$$U_0^2 \mathrel{\substack{30 \\ b\longrightarrow \\ \longleftarrow f \\ 19}} U_0^3 \mathrel{\substack{13 \\ b\longrightarrow \\ \longleftarrow f \\ 0}} U_0^1 \mathrel{\substack{0 \\ b\longrightarrow \\ \longleftarrow f \\ 0}} U_1^2 \mathrel{\substack{32 \\ b\longrightarrow \\ \longleftarrow f \\ 18}} U_1^1 \mathrel{\substack{18 \\ b\longrightarrow \\ \longleftarrow f \\ 9}} U_1^3 \cdots \tag{1}$$

Because U_0^1 and U_1^2 are in the concurrent relation, the forward gap and backward gap time are set to 0.

A.4. Correctness of the second phase

In this section, we would like to prove that the DODA algorithm achieves the goal of constructing a unique order view for all video servers. We use three lemmas to complete our proof.

Lemma 1. *For any server k, given any two service units U_i and U_j with U_i leading U_j, if the setup message sm_j^x for U_j arrives at server k at time $TA(sm_j^x)$, then the setup message sm_i^y for U_i will arrive before $TA(sm_j^x)$ or within $[TA(sm_j^x), TA(sm_j^x) + T_{\text{maxDelay}}]$.*

Proof: The result is very straightforward. Let us consider two extreme cases. In the first, sm_i^y arrives at server k using minimal transmission time T_{minDelay} and sm_j^x using maximal transmission time T_{maxDelay}. Obviously, sm_i^y arrives at server k before the arrival of sm_j^x. In contrast, sm_i^y arrives at server k using maximal transmission time T_{maxDelay} and sm_j^x using minimal transmission time T_{minDelay}. If sm_j^x arrives at server k first and the sm_i^y arrives after $TA(sm_j^x)$ plus T_{maxDelay}. The time instant for server y to broadcast sm_i^y should be after $TA(sm_j^x)$ due to the use of the maximal transmission delay. The result is in conflict with the assumption of U_i leading U_j. □

Table 1. Message flow.

Server 1	Server 2	Server 3
16: Rc. (2, S, V[Nil, 0, Nil]$_{2}$, F[−1, −1, −1], B[−1, −1, −1])	3: Br. (2, S, V[Nil, 0, Nil]$_{2}$, F[−1, −1, −1], B[−1, −1, −1])	18: Rc. (2, S, V[Nil, 0, Nil]$_{2}$, F[−1, −1, −1], B[−1, −1, −1])
36: Br. (1, S, V[0, 0, Nil]$_{1}$, F[−1, 26, −1], B[−1, −1, −1])	34: Br. (2, S, V[Nil, 1, Nil]$_{2}$, F[−1, 31, −1], B[−1, 31, −1])	31: Br. (3, S, V[Nil, 0, 0]$_{3}$, F[−1, 19, −1], B[−1, −1, −1])
43: Rc. (3, S, V[Nil, 0, 0,]$_{3}$, F[−1, 19, −1], B[−1, −1, −1])	39: Rc. (3, S, V[Nil, 0, 0]$_{3}$, F[−1, 19, −1], B[−1, −1, −1]) then Br. (2, M_{SLR}, V[Nil, 1, 0]$_{2}$, F[−1, 31, 1], B[−1, 31, −1]) and Br. (2, M, V[Nil, 0, 0]$_{3}$, F[−1, 19, −1], B[−1, 30, −1])	43: Rc. (2, S, V[Nil, 1, Nil]$_{2}$, F[−1, 31, −1], B[−1, 31, −1])
47: Rc. (2, S, V[Nil, 1, Nil]$_{2}$, F[−1, 31, −1), B[−1, 31, −1])	47: Rc. (1, S, V[0, 0, Nil]$_{1}$, F[−1, 26, −1], B[−1, 38, −1]) then Br. (2, M, V[0, 0, Nil]$_{1}$, F[−1, 26, −1], B[−1, 38, −1])	50: Rc. (1, S, V[0, 0, Nil]$_{1}$, F[−1, 26, −1], B[−1, −1, −1]) then Br. (3, M_{RLS}, V[0, 0, 0]$_{1}$, F[−1, 26, −1], B[−1, −1, 13])
52: Rc. (2, M_{SLR}, V[Nil, 1, 0]$_{2}$, F[−1, 31, 1), B[−1, 31, −1]) and Rc. (2, M, V[Nil, 0, 0]$_{3}$, F[−1, 19, −1], B[−1, 30, −1])	58: Rc. (3, M_{RLS}, V[0, 0, 0]$_{1}$, F[−1, 26, −1], B[−1, −1, 13])	55: Rc. (2, M, V[Nil, 2, 0]$_{2}$, F[−1, 31, 1], B[−1, 31, −1]) and Rc. (2, M, V[Nil, 0, 0]$_{3}$, F[−1, 19, −1], B[−1, 30, −1])
56: Rc. (2, M, V[0, 0, Nil]$_{1}$, F[−1, 26, −1], B[−1, 38, −1])	63: Rc. (3, M, V[Nil, 1, 0]$_{2}$, F[−1, 31, 1], B[−1, 31, 6])	58: Rc. [2, M, V[0, 0, Nil]$_{1}$, F[−1, 26, −1], B[−1, 38, −1])
58: Rc. (3, M_{RLS}, V[0, 0, 0]$_{1}$, F[−1, 26, −1], B[−1, −1, 13]) then Br. (1, M, V[0, 0, 0]$_{1}$, F[−1, 26, 0], B[−1, −1, 13])	67: Rc. (1, M, V[0, 0, 0]$_{1}$, F[−1, 26, 0], B[−1, −1, 13])	65: Rc. (1, M, V[0, 0, 0]$_{1}$, F[−1, 26, 0], B[−1, −1, 13])
59: Br. (1, S, V[1, 1, 0]$_{1}$, F[23, 18, 22], B[23, −1, −1])	72: Rc. (1, S, V[1, 1, 0]$_{1}$, F[23, 18, 22], B[23, −1, −1]) then Br. (2, M, V[1, 1, 0]$_{1}$, F[23, 18, 22], B[23, 32, −1])	69: Rc. (1, S, V[1, 1, 0]$_{1}$, F[23, 18, 22], B[23, −1, 32]) then Br. (3, S, V[1, 1, 0]$_{1}$, F[23, 18, 22], B[23, −1, −1])
64: Rc. (3, M, V[Nil, 1, 0]$_{2}$, F[−1, 31, 1], B[−1, 31, 6])	81: Rc. (3, M, V[1, 1, 0]$_{1}$, F[23, 18, 22], B[23, −1, 32])	72: Br. (2, S, V[1, 1, 1]$_{3}$, F[9, 35, 41], B[−1, −1, 41])
80: Rc. (2, M, V[1, 1, 0]$_{1}$, F[23, 18, 22], B[23, 32, −1])	84: Rc. (3, S, V[1, 1, 1]$_{3}$, F[9, 35, 41], B[−1, −1, 41]) then Br. (2, M, V[1, 1, 1]$_{3}$, F[9, 35, 41], B[−1, 44, 41])	82: Rc. (2, M, V[1, 1, 0]$_{1}$, F[23, 18, 22], B[23, 32, −1])
83: Rc. (3, S, V[1, 1, 1]$_{3}$, F[9, 35, 41], B[−1, −1, 41]) then Br. (1, M, V[1, 1, 1]$_{3}$, F[9, 35, 41], B[18, −1, 41]		94: Rc. (2, M, V[1, 1, 1,]$_{3}$, F[9, 35, 41], B[−1, 44, 41])
86: Rc. (3, M, V[1, 1, 0]$_{1}$, F[23, 18, 22], B[23, −1, 32])		96: Rc. (1, M, V[1, 1, 1]$_{3}$, F[9, 35, 41], B[18, −1, 41])
92: Rc. (2, M, V[1, 1, 1]$_{3}$, F[9, 35, 41], B[−1, 44, 41])		

Table 2. TOL, ODL, and STL transitions.

```
(0)
TOL: Nil
ODL: Nil
STL: Nil
$S_1^1 = -1$, $S_2^1 = -1$, $S_3^1 = -1$

(16)
TOL: Nil
ODL: Nil
STL: TA(16), V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1] (61)
$S_1^1 = -1$, $S_2^1 = 0$, $S_3^1 = -1$

(36)
TOL: Nil
ODL: Nil
STL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1] (61)
     TA(36) V[0, 0, Nil]$_1$ F[-1, 26, -1] B[-1, -1, -1] (87)
$S_1^1 = 0$, $S_2^1 = 0$, $S_3^1 = -1$

(43)
TOL: Nil
ODL: Nil
STL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1] (61)
     TA(36) V[0, 0, Nil]$_1$ F[-1, 26, -1] B[-1, -1, -1] (87)
     TA(43) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, -1, -1] (88)
$S_1^1 = 0$, $S_2^1 = 0$, $S_3^1 = 0$

(47)
TOL: Nil
ODL: Nil
STL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1] (61)
     TA(36) V[0, 0, Nil]$_1$ F[-1, 26, -1] B[-1, -1, -1] (87)
     TA(43) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, -1, -1] (88)
     TA(47) V[Nil, 1, Nil]$_2$ F[-1, 31, -1] B[-1, 31, -1] (92)
$S_1^1 = 0$, $S_2^1 = 1$, $S_3^1 = 0$

(52)
TOL: Nil
ODL: Nil
STL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1] (61)
     TA(36) V[0, 0, Nil]$_1$ F[-1, 26, -1] B[-1, -1, -1] (87)
     TA(43) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, -1] (92)
$S_1^1 = 0$, $S_2^1 = 1$, $S_3^1 = 0$

(56)
TOL: Nil
ODL: Nil
STL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1] (61)
     TA(36) V[0, 0, Nil]$_1$ F[-1, 26, -1] B[-1, 38, -1] (87)
     TA(43) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, -1] (92)
$S_1^1 = 0$, $S_2^1 = 1$, $S_3^1 = 0$
```

(*Continued on next page.*)

Table 2. (*Continued*).

```
(58)
TOL: Nil
ODL: Nil
STL: TA(16) V[Nil, 0, Nil]_2 F[-1, -1, -1] B[-1, -1, -1] (61)
     TA(36) V[0, 0, 0]_1 F[-1, 26, 0] B[-1, 38, 13] (87)
     TA(43) V[Nil, 0, 0]_3 F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]_2 F[-1, 31, 1] B[-1, 31, -1] (92)
S_1^1= 0, S_2^1= 1, S_3^1= 0

(59)
TOL: Nil
ODL: Nil
STL: TA(16) V[Nil, 0, Nil]_2 F[-1, -1, -1] B[-1, -1, -1] (61)
     TA(36) V[0, 0, 0]_1 F[-1, 26, 0] B[-1, 38, 13] (87)
     TA(43) V[Nil, 0, 0]_3 F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]_2 F[-1, 31, 1] B[-1, 31, -1] (92)
     TA(59) V[1, 1, 0]_1 F[23, 18, 22] B[23, -1, -1] (104)
S_1^1= 1, S_2^1= 1, S_3^1= 0

(61)
TOL: Nil
ODL: TA(16) V[Nil, 0, Nil]_2 F[-1, -1, -1] B[-1, -1, -1] (78)
STL: TA(36) V[0, 0, 0]_1 F[-1, 26, 0] B[-1, 38, 13] (87)
     TA(43) V[Nil, 0, 0]_3 F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]_2 F[-1, 31, 1] B[-1, 31, -1] (92)
     TA(59) V[1, 1, 0]_1 F[23, 18, 22] B[23, -1, -1] (104)
S_1^1= 1, S_2^1= 1, S_3^1= 0

(64)
TOL: Nil
STL: TA(16) V[Nil, 0, Nil]_2 F[-1, -1, -1] B[-1, -1, -1] (78)
ODL: TA(36) V[0, 0, 0]_1 F[-1, 26, 0] B[-1, 38, 13] (87)
     TA(43) V[Nil, 0, 0]_3 F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]_2 F[-1, 31, 1] B[-1, 31, 6] (92)
     TA(59) V[1, 1, 0]_1 F[23, 18, 22] B[23, -1, -1] (104)
S_1^1= 1, S_2^1= 1, S_3^1= 0

(78)
TOL: TA(16) V[Nil, 0, Nil]_2 F[-1, -1, -1] B[-1, -1, -1]
ODL: Nil
STL: TA(36) V[0, 0, 0]_1 F[-1, 26, 0] B[-1, 38, 13] (87)
     TA(43) V[Nil, 0, 0]_3 F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]_2 F[-1, 31, 1] B[-1, 31, 6] (92)
     TA(59) V[1, 1, 0]_1 F[23, 18, 22] B[23, -1, -1] (104)
S_1^1= 1, S_2^1= 1, S_3^1= 0

(80)
TOL: TA(16) V[Nil, 0, Nil]_2 F[-1, -1, -1] B[-1, -1, -1]
ODL: Nil
STL: TA(36) V[0, 0, 0]_1 F[-1, 26, 0] B[-1, 38, 13] (87)
     TA(43) V[Nil, 0, 0]_3 F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]_2 F[-1, 31, 1] B[-1, 31, 6] (92)
     TA(59) V[1, 1, 0]_1 F[23, 18, 22] B[23, 32, -1] (110)
S_1^1= 1, S_2^1= 1, S_3^1= 0
```

(*Continued on next page.*)

Table 2. (*Continued*).

```
(83)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1]
ODL: Nil
STL: TA(36) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13] (87)
     TA(43) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6] (92)
     TA(59) V[1, 1, 0]$_1$ F[23, 18, 22] B[23, 32, -1] (110)
     TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, -1, 41] (128)
$S_1^1$= 1, $S_2^1$= 1, $S_3^1$= 1

(86)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1]
ODL: Nil
STL: TA(36) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13] (87)
     TA(43) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6] (92)
     TA(59) V[1, 1, 0]$_1$ F[23, 18, 22] B[23, 32, 32] (110)
     TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, -1, 41] (128)
$S_1^1$= 1, $S_2^1$= 1, $S_3^1$= 1

(87)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1) B(-1, -1, -1)
ODL: TA(36) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13] (104)
STL: TA(43) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1] (88)
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6] (92)
     TA(59) V[1, 1, 0]$_1$ F[23, 18, 22] B[23, 32, 32] (110)
     TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, -1, 41] (128)
$S_1^1$= 1, $S_2^1$= 1, $S_3^1$= 1

(88)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1]
ODL: TA(36) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13] (104)
     TA(43) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1] (105)
STL: TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6] (92)
     TA(59) V[1, 1, 0]$_1$ F[23, 18, 22] B[23, 32, 32] (110)
     TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, -1, 41] (128)
$S_1^1$= 1, $S_2^1$= 1, $S_3^1$= 1

(92)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1]
ODT: TA(36) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13] (104)
     TA(43) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1] (105)
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6] (109)
STL: TA(59) V[1, 1, 0]$_1$ F[23, 18, 22] B[23, 32, 32] (110)
     TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, 44, 41] (128)
$S_1^1$= 1, $S_2^1$= 1, $S_3^1$= 1

(104)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1]
     TA(36) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1]
     TA(43) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13]
ODT: TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6] (109)
STT: TA(59) V[1, 1, 0]$_1$ F[23, 18, 22] B[23, 32, 32] (110)
     TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, 44, 41] (128)
$S_1^1$= 1, $S_2^1$= 1, $S_3^1$= 1
```

(*Continued on next page.*)

Table 2. (*Continued*).

```
(109)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1]
     TA(36) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1]
     TA(43) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13]
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6]
ODL: Nil
STL: TA(59) V[1, 1, 0]$_1$ F[23, 18, 22 B[23, 32, 32] (110)
     TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, 44, 41] (128)
$S_1^1 = 1$, $S_2^1 = 1$, $S_3^1 = 1$

(110)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1]
     TA(36) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1]
     TA(43) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13]
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6]
ODL: TA(59) V[1, 1, 0]$_1$ F[23, 18, 22] B[23, 32, 32] (127)
STL: TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, 44, 41] (128)
$S_1^1 = 1$, $S_2^1 = 1$, $S_3^1 = 1$

(127)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1]
     TA(36) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1]
     TA(43) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13]
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6]
     TA(59) V[1, 1, 0]$_1$ F[23, 18, 22] B[23, 32, 32]
ODL: Nil
STL: TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, 44, 41] (128)
$S_1^1 = 1$, $S_2^1 = 1$, $S_3^1 = 1$

(128)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1]
     TA(36) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1]
     TA(43) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13]
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6]
     TA(59) V[1, 1, 0]$_1$ F[23, 18, 22] B[23, 32, 32]
ODL: TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, 44, 41] (135)
STL: Nil
$S_1^1 = 1$, $S_2^1 = 1$, $S_3^1 = 1$

(135)
TOL: TA(16) V[Nil, 0, Nil]$_2$ F[-1, -1, -1] B[-1, -1, -1]
     TA(36) V[Nil, 0, 0]$_3$ F[-1, 19, -1] B[-1, 30, -1]
     TA(43) V[0, 0, 0]$_1$ F[-1, 26, 0] B[-1, 38, 13]
     TA(47) V[Nil, 1, 0]$_2$ F[-1, 31, 1] B[-1, 31, 6]
     TA(59) V[1, 1, 0]$_1$ F[23, 18, 22] B[23, 32, 32]
     TA(83) V[1, 1, 1]$_3$ F[9, 35, 41] B[18, 44, 41]
ODL: Nil
STL: Nil
$S_1^1 = 1$, $S_2^1 = 1$, $S_3^1 = 1$
```

Lemma 2. *For any server k, given any two service units U_i and U_j with an concurrent relationship, the distance between the arrival time sm_i^y and sm_j^x at server k is not longer than T_{maxDelay}, namely, $|TA(sm_i^y) - TA(sm_j^x)| < T_{\text{maxDelay}}$.*

Proof: This proof is similar to that for Lemma 1. □

Lemma 3. *If two strings are identical, given any element, say a, in these two strings. The preceding elements to a in both strings will be the same regardless of the order of elements. In contrast, given two strings, if the elements preceding an element, say a, on both strings are the same, then these two strings are identical regardless of the order of elements.*

Using Lemma 3, we consider the order interpreted from the total order lists as strings. If we can prove that the preceding service units to any given service unit are the same in all TOLs, then the TOLs in all the servers are *identical*.

Theorem 1. *The TOLs constructed by the distributed order decision algorithm are identical for all servers.*

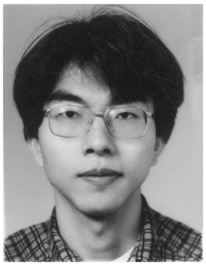
Proof: Given two servers, say Server 1 and server 2, their TOLs are l_1 and l_2, respectively. For any two members in l_1, say b_x and b_y, assuming that b_y's order goes ahead of b_x in l_1, then we should prove that the same order relationship exists in l_2. We now discuss the three possible relationships for b_x and b_y. In the first case, both order vectors for b_x and b_y indicate b_y leading b_x. With Lemma 1, if the arrival pattern is b_x before b_y, b_x and b_y will be handled together in the order-decision procedure invoked by b_x's expired timer. When b_x becomes pop-able in l_2, b_y must already be in l_2 or b_x will have no chance to become pop-able. In the second case, b_x and b_y are in a concurrent relationship but the server's identification of b_x is greater than that of b_y. These two elements will be handled by either of their timer-expired procedures according to Lemma 2. Because b_x and b_y are confused with each other, they become pop-able simultaneously. However, they will be sorted by their servers' identification and then b_y will be popped first due to the smaller server identification. In the final case, b_x and b_y are also in confusion but the server's identification of b_x is smaller than that of b_y. There must be an element b_z such that b_y is leading b_z, b_z is leading b_x but b_x is confused with b_y. Though b_y is confused with b_x and b_y's server identification is greater than b_x, b_y becomes pop-able first but the b_x is still not pop-able due to the b_z is not in l_2. □

Acknowledgments

We thank the referees for their comments and suggestions. This work was partially sponsored by the National Science Council, Taiwan, R.O.C. under contract number: NSC85-2221-E-009-039.

References

1. J.W. Byun and T.T. Lee, “The design and analysis of an ATM multicast switch with adaptive traffic controller,” IEEE/ACM Trans. on Networking, Vol. 2, No. 3, pp. 288–298, 1994.
2. M.S. Chen, D.D. Kandlur, and P.S. Yu, “Storage and retrieval methods to support fully interactive playout in a disk-array-based video server,” Multimedia Systems, Vol. 3, pp. 126–135, 1995.
3. F. Cristian, “Understanding fault-tolerant distributed systems,” Comm. of the ACM, Vol. 34, pp. 56–78, 1991.
4. A. Dan, D. Sitaram, and P. Shahabuddin, “Dynamic batching policies for an on-demand video server,” Multimedia Systems, Vol. 4, No. 3, pp. 112–121, 1996.
5. W. Effelsberg and T. Haerder, “Principles of database buffer management,” ACM Trans. on Database Systems, Vol. 9, No. 4, pp. 560–595, 1984.
6. C. Federighi and L.A. Rowe, “A distributed hierarchical storage manager for a video-on-demand system,” in Proc. of IS&T/SPIE, San Jose, CA, 1994.
7. E.A. Fox, “The coming revolution of interactive digital video,” Comm. of the ACM, Vol. 32, pp. 794–801, 1989.
8. E.A. Fox, “Standards and emergence of digital multimedia systems,” Comm. of the ACM, Vol. 34, pp. 26–30, 1991.
9. B. Furht, “Multimedia systems: An overview,” IEEE Multimedia, pp. 47–59, 1994.
10. D. Le Gall, “MPEG: A video compression standard for multimedia applications,” Comm. of the ACM, Vol. 34, No. 4, pp. 46–58, 1991.
11. E. Gelenbe, D. Finkel, and S. Tripathi, “Availability of a distributed computer system with failures,” Acta Informatica, Vol. 23, pp. 643–655, 1986.
12. D.J. Gemmell, “Multimedia storage servers: A tutorial,” IEEE Multimedia, pp. 40–49, 1995.
13. L. Golubchik, C.S. Lui, and R. Muntz, “Adaptive piggybacking: A novel technique for data sharing in video-on-demand storage servers,” Multimedia Systems, Vol. 4, No. 3, pp. 140–155, 1996.
14. J. Huang and L.A. Stankovic, “Buffer management in real-time database,” Department of Computer Science, University of Massachusetts, Technique Report, pp. 90–65, July 1990.
15. Y. Huang and S.K. Tripathi, “Resource allocation for primary-site fault-tolerant systems,” IEEE Trans. on Software Engineering, Vol. 19, No. 2, 1993.
16. J.C. Laprie, J. Arlat, and C. Beounes, “Definition and analysis of hardware- and software-fault-tolerant architectures,” IEEE Computer, Vol. 23, pp. 39–51, 1990.
17. T.D.C. Little and D. Venkatesh, “Popularity-based assignment of movies to storage devices in a video-on-demand system,” Multimedia Systems, Vol. 2, No. 6, pp. 280–2987, 1995.
18. K. Nahrstedt, “Resource management in networked multimedia systems,” IEEE Multimedia, pp. 52–63, 1995.
19. V.P. Nelson, “Fault-tolerant computing: Fundamental Concepts,” IEEE Computer, Vol. 23, pp. 19–25, 1990.
20. K.K. Ramakrishnan, “Operating system support for a video-on-demand file service,” Multimedia Systems, Vol. 3, pp. 53–63, 1995.
21. P.V. Rangan, H.M. Vin, and S. Ramanathan, “Designing an on-demand multimedia service,” IEEE Communications, Vol. 30, pp. 56–64, 1992.
22. D. Rotem and J.L. Zhao, “Buffer management for video database systems,” IEEE Intl. Conf. on Data Engineering, 1995, pp. 439–448.
23. R.D. Schlichting and F.B. Schneider, “Fail-stop processors: An approach to designing fault-tolerant computing systems,” ACM Transactions on Computing Systems, Vol. 1, pp. 222–238, 1993.
24. F.B. Schneider, “Byzantine generals in action: Implementing fail-stop processors,” ACM Trans. on Computing Systems, Vol. 2, pp. 145–154, 1984.
25. D.P. Siewiorek, “Fault tolerance in commercial computers,” IEEE Computer, Vol. 23, pp. 19–25, 1990.
26. G.K. Zipf, Human Behavior and the Principles of Least Effort, Addison-Wesley: Reading, MA, 1949.



Ing-Jye Shyu received his B.S. degree in Computer Science and Information Engineering from the National Chiao-Tung University, Taiwan, in 1992. He is currently a Ph.D. candidate at the same university. His research interests include video-on-demand systems, distributed communication protocols, and real-time operating systems.

Shiuh-Pyng Shieh received the M.S. and Ph.D. degrees in Electrical Engineering from the University of Maryland, College Park, in 1986 and 1991, respectively. He is currently a Professor with the Department of Computer Science and Information Engineering, National Chiao-Tung University. From 1988 to 1991 he participated in the design and implementation of the B2 Secure XENIX for IBM, Federal Sector Division, Gaithersburg, Maryland, USA. He is also the designer of SNP (Secure Network Protocol). Since 1994 he has been a consultant for Computer and Communications Laboratory, Industrial Technology Research Institute, Taiwan in the area of network security and distribute operating systems. He is also a consultant for the National Security Bureau, Taiwan. Dr. Shieh was on the organizing committees of a number of conferences, such as International Computer Symposium, and International Conference on Parallel and Distributed Systems. Recently, he is the program chair of 1997 Information Security Conference (INFOSEC ’97). His research interests include distributed operating systems, and computer security.