Redundant Array of Inexpensive Servers (RAIS) for On-demand Multimedia Services

P.C. Wong and Y.B. Lee
Advanced Network Systems Laboratory
Department of Information Engineering
The Chinese University of Hong Kong, Hong Kong
{pcwong, yblee}@ie.cuhk.edu.hk

Abstract

In [1], we considered the design and implementation of a server array system for delivering video-on-demand service on a computer network. Data blocks are striped across an array of autonomous servers, and each client station contacts the servers one by one to retrieve blocks for constructing its own video stream. The server array approach has the benefits of (1) scalable storage capacity and throughput, and (2) load sharing across the servers. In this paper, we study fault-tolerance issues in a server array system and propose the notion of *Redundant Array of Inexpensive Servers* (*RAIS*), which is a server-level counterpart to *Redundant Array of Inexpensive Disks* (*RAID*). While some concepts from RAID may be applicable, we show that RAIS faces new challenges as failure detection and recovery are handled by network protocols, subjecting to error, loss, non-deterministic delay, and bandwidth limitation. Through an implementation of RAIS for delivering multimedia world-wide-web services, we show that (1) server-level fault tolerance can be achieved, (2) continuous playback of video and audio can be maintained in spite of server failures, and (3) graceful service degradation can be implemented using our Object Striping scheme.

1. Introduction

With the exploding success of the world-wide-web (WWW), multimedia services (text, graphics, video, audio, etc.) are becoming common place in both internet and intranet applications. The ease of retrieving multimedia information on-line using a client-server model becomes the norm of most applications, and the design of network systems and servers for delivering on-demand multimedia services becomes a hot topic of research.

Nearly all of today's servers run on a single machine connected to the network. To achieve a higher server capacity, one may use server clusters and distribute requests to individual servers for a particular piece of information - *Service partitioning*. This technique allows the loading to be shared by several servers, but if many clients are accessing one particular server for a specific object (e.g., a popular video), the *hot-spot* server will still be overloaded. On the other hand, server failures will render a subset of data inaccessible. Another technique is to replicate data on several servers - *Service replication*. This results in management complexity, and the storage needed will be several times of the original data. Given that multimedia objects like good quality video require vast amount of storage (e.g., a 90-minute MPEG 1 movie requires more than 1 GByte), service replication clearly has its limitation.

In [1], we considered a server array for delivering video-on-demand service on a local area network. Video blocks are striped across an array of servers, and each client contacts the servers one by one to retrieve the blocks for its own video stream - *Service striping*. In this paper, we propose the notion of *Redundant Array of Inexpensive Servers* (RAIS) architecture such that server-level fault tolerance can be achieved. Like RAID, a RAIS system achieves a scalable system capacity and load sharing among servers. With server-level fault-tolerant capability, the system can maintain a continuous service (e.g., non-stop service to video and data streams) to client stations despite one or more servers fail.

2. Redundant Array of Inexpensive Servers (RAIS)

Figure 1 shows the network architecture of a general server array system. Data blocks of each stream are striped across the array of servers, for example block 1 on server 1, block 2 on server 2, and so on. A fast packet switch is used to connect server and client stations. Each server has a dedicated network segment, and each client contacts the server one by one for retrieving blocks of a particular stream, and reconstructs back its own stream. Since each server has its own storage, CPU, and network segment, the overall server capacity increases with the number of servers. Furthermore, one may increase the capacity at any time simply by adding one more server and redistributing the data over the servers.

With an increased number of components, reliability becomes an issue. In disk array, the increased number of disk drives reduced the overall reliability as any disk failure can render the entire disk array unusable. The solution in disk array is to introduce error-correcting coding and store redundant data to support fault tolerance - Redundant Array of Inexpensive Disks (RAID) [2]. Similar problem also exists in a server array and we propose extending the RAID concept to the server level, forming a Redundant Array of Inexpensive Servers (RAIS).

While RAID and RAIS are similar in concept, we find that most RAID configurations are not feasible to work at the server level. First, RAID-2, and RAID-3 stripe data on bit and byte levels. Since data is reassembled at the client, probably using a software module, reassembling multiple interleaved bit or byte streams into a single stream would require too much processing. Secondly, we may want to use different stripe size

for different media type to optimize the I/O efficiency. Finally, we may want to assign different levels of redundancy to different media type to allow graceful service degradation during server failures. All these cannot be supported using the RAID schemes. We propose in the following an Object Striping scheme for RAIS to meet all these requirements.

3. Object Striping

Object striping does striping at the data object level. In other words, striping is performed on each individual object, such as a text page, an image, audio, or video file.

We propose a *Dynamic Object Striping* (DOS) scheme to optimize striping for various kinds of media data. In DOS, all storage is divided into small stripe units (e.g. 1KB), called *micro-blocks*, and objects can be represented as a multiple of micro-blocks. Using small stripe units solves the problem of internal fragmentation. Under DOS, each object has its own striping policy consisting of three attributes:

{*START_UNIT, UNIT_SIZE, REDUNDANCY*}

The DOS policy is created whenever an object is created and stored into the server array. The DOS policy may be changed if necessary, but the data will have to be redistributed over the servers. START_UNIT records the starting stripe unit of the particular data object; UNIT_SIZE records the number of micro-blocks of that object to be striped on one server as a *macro-block*. In this way, each object is striped across the servers in terms of macro-blocks instead of micro-blocks, and different objects may have a different size of macro-block (i.e., stripe unit size). REDUNDANCY records the level of redundancy employed in storing the object. A larger value of REDUNDANCY means a higher level of redundancy.

We see that different types of objects can be striped in a different manner. Figure 2 shows that large objects can be assigned DOS policy with large UNIT_SIZE (e.g. 64 micro-blocks for video streams) for optimizing the disk I/O efficiency. Each client request will retrieve 64 micro-blocks from one server at a time. Smaller objects or infrequently accessed data objects can be assigned a DOS policy with small UNIT_SIZE (e.g. one or two micro-blocks). On the other hand, video objects can be assigned REDUNDANCY of 1, graphics with REDUNDANCY of 2, and texts with REDUNDANCY of 3. In this way, video service survive single-server failures, whereas graphics and texts will survive double and triple-server failures respectively. This flexibility allows multimedia services to degrade gracefully under multiple server failures.

4. Failure Detection and Recovery

Analogous to RAID, we classify RAIS operation into five modes [3]: *normal mode, failure mode, reconstruction mode, reconfigured mode, and restoration mode*. In normal mode, all servers are operational. The system enters failure mode once a server failure is detected. Redundant blocks are retrieved by the client for lost data recovery. The system then commences the reconstruction mode to reconstruct the lost data of the failed server into a spare server. When the reconstruction is completed, the system enters the reconfigured mode in which the spare server replaces the failed server. When the failed server is replaced, the system begins restoration mode to bring the system back into normal configuration. In this paper, we focus only on how a client can recover the lost data blocks to maintain a continuous stream service. Interested readers are referred to [3] for details on lost-data reconstruction.

4.1 Lost Data Recovery

When a server fails, the client needs redundant blocks from the remaining operating servers to recover the lost blocks at the failed server. In this respect, we propose two possible approaches: *Forward-Erasure-Correction* (FEC) and *On-Demand-Correction* (ODC).

Under FEC, redundant data are always transmitted to the clients, even when no server fails. As a client always receives the redundant data, no actions need to be taken during server failure. The downside is increased system utilization during normal-mode operation. Specifically, the extra overhead in server, network, and client bandwidth is $k/(N-k)$, where N is the number of stripe units in each stripe for that object, and k is the redundancy level for that object. For example, with a (15,11) RS-code, we have N=15, k=4, the redundancy overhead is 4/11 = 36%. To reduce this overhead, we propose a scheme called *Progressive Redundancy Transmission* (PRT).

In PRT, we take advantage of the fact that a server fails with very low probability, and the probability of two or more servers failing in a short time interval is negligible. During normal mode, we transmit only $(N-k+1)$ out of the N symbols in the $(N, N-k)$ code. When one server fails, all data symbols can still be recovered from the remaining $(N-k)$ symbols. At the same time, we start transmitting one more symbol along with the remaining $(N-k)$ symbols, resulting in again $(N-k+1)$ symbols. The system can then tolerate another server failure and the process repeats again. The server, network, and client overhead will be reduced to $1/(N-k)$ instead of $k/(N-k)$. In the (15,11) RS code example, this translates into only 9% overhead compared to the original 36% overhead.

We can generalize the above PRT scheme to transmit $(N-k+r)$ symbols at normal operation, where $0 \le r \le (N-k)$. Then the system will be able to tolerate at most r simultaneous server failures. In this case, the transmission overhead will be $r/(N-k)$. Therefore the PRT scheme allows a system designer to trade-off between redundancy overhead and system reliability.

Under On-Demand-Correction (ODC), redundant data are not transmitted during normal-mode operation. When a client detects a server failure, it initiates failure-mode operation and starts requesting redundant data to reconstruct the lost data. This approach does not require extra bandwidth during normal-mode operation but increases the delay for receiving a block when a server fails. We study this delay in the following and derive the buffer size required for continuous- playback of stream-type services.

4.2 Continuity in Stream-type Services

For stream-type services like video and audio, the additional system delay means extra client buffer requirement to maintain service continuity. We consider a simple credit-based model to illustrate this additional client buffer requirement. In this model, the client has N_B blocks of buffer prefilled with data before playback commences. Whenever the client finishes using one block from the buffer for playback, it submits a new request to a server for a new block to fill up the emptied buffer. In this way, the client buffer will never be overflown. If the client can always receive a block before it empties its buffer, the stream will maintain its continuity.

We assume that the server *response* time (i.e. time from sending a request to the server to the time a complete response is received) is bounded by an upper limit D_{max}. Figure 3 shows an empirical server response time distribution and such a bound. We further assume that requests are generated quasi-periodically with a minimum period of T seconds.

First consider the case of no server failure. We see that the request time is coupled with the playback time for each block. Let T be the minimum playback time of any block of the stream, which is also the minimum time interval between requests. When a client begins to play the $(k+N_B-1)^{th}$ block, the request of this block must have been submitted (N_B-1) block times earlier, i.e., while the client was starting to play the kth video block. In order that the block is available before its playback, we must have

$$(N_B - 1)\, T \geq D_{max} \qquad (1)$$

or

$$N_B \geq (D_{max}/T) + 1 \qquad (2)$$

which is the client buffer needed to ensure that all data blocks will be received in time for playback.

When one or more servers fail, we need extra buffers for (i) storing at least a full stripe of units for erasure-correction and (ii) absorbing the additional delay for retrieving the parity units.

For a $(N, N-k)$ RS-code encoded data stripe, at least $(N-k)$ out of N stripe units are required to reconstruct the original data. Hence we need to completely receive the first $(N-k)$ stripe units in the prefetch buffer every time a stripe unit is consumed for playback. Let n be the number of buffer units used for storing data stripe units, then we can extend Equation (1) as follows:

$$(n - (N - k))\, T \geq D_{max} \qquad (3)$$

However, we have not included buffers for storing incoming parity units. Under PRT, r parity units are transmitted along with data units, therefore we need a total of

$$N_B \geq n + \lceil n/(N-k) \rceil r \qquad (4)$$

buffer units to maintain stream continuity under the FEC scheme.

In the ODC case, the worst-case failure-detection time occurs when the server fails immediately after acknowledging a request packet without servicing it. This scenario is depicted in Figure 4. Under this worst-case scenario the server failure will be detected when retransmission requests are sent to the failed server after a time D_{max}. Incorporating this into Equation (1), we have the condition for the lost stripe unit to arrive on-time:

$$(N_B - (N - k))\, T \geq D_{max} + D_{max} + T_{DETECT} \qquad (5)$$

or

$$N_B \geq (2D_{max} + T_{DETECT})/T + (N - k) \qquad (6)$$

5. Implementation Results

To demonstrate the feasibility of the RAIS architecture, we developed a *WebArray* system to deliver on-demand multimedia world-wide-web services. Alongside of WWW requests, the system supports high quality video (MPEG 1) and audio streams. The system runs on Pentium-90 PCs with Windows NT 4.0 operating system. Figure 5 depicts the access to our WebArray using a WWW browser (Netscape Navigator), displaying text, graphics, and streaming video simultaneously.

To ensure that video and audio streams can maintain continuity irrespective of data traffic, we implemented scheduling algorithms at both the disk and the network to give a higher priority to stream-type traffic. WWW requests are generated by traffic generators on client PCs based on WebStone 2.0 [5] and the standard test-file size distribution shown in Figure 6. Requests for video data are generated by actual video connections. Each client PC establishes an active video connection for viewing a 1.2Mbps MPEG-1 video, which is a 30 fps, near TV quality video through a web page. The detailed design of this project is given in [4]. Here, we focus our results on the server capacity and fault tolerant capability.

5.1 RAIS Capacity and Scalability

We benchmarked the WWW service and the video service separately. The results are shown in Figure 7, comparing the WWW and video capacities versus the number of servers. We see that the capacity in both cases scales up with the number of servers almost in a linear fashion. This demonstrated the scalability of the RAIS architecture.

On the other hand, video service achieves higher capacity than WWW service under the same configuration. This is because video objects are significantly larger than data objects. As a result, video service requires significantly less connection and delivery overhead.

5.2 RAIS Server Utilization During Normal and Failure Mode

We conduct experiments to obtain the server utilization before and after a server failure. The test is run with four servers using distributed single-parity coding. In all cases, client video continues playback without interruption during and after server failure.

Figure 8 shows the test result for four servers with 10 concurrent video sessions. Fault tolerance is supported using On-Demand-Correction (ODC). Server 3 is terminated (disruptively) after 3 minutes. The fault is detected by all clients and subsequently all clients enter failure-mode operation. The extra retrieval of parity data results in

increases in server throughput after the fault. In this test, transmission overhead for parity data is 1/(4-1) or 33%, which agrees with our results.

Figure 9 shows a similar test where we transmit redundant data continuously even under normal operating mode (Forward-Erasure-Correction). Same as the pervious case, Server 3 is terminated (disruptively) after 3 minutes. While the fault is detected by the clients, no special action needs to be taken because parity data are always transmitted. Hence there is no increase in server throughput after failure. Rather, the transmission overhead (33%) exists before and after server failure.

The above tests show that our RAIS system can perform lost data recovery and maintain video service continuity in the midst of server failures. We also see that by using On-Demand-Correction, there is no need to transmit redundant data during normal operating condition. Rather, redundant data transmissions are needed only after a server failure is detected. Consequently, the system loading during normal operating condition is lower, giving a better delay response to all services before a server failure which is supposed to be a rare case.

5.3 RAIS Client Utilization During Normal and Failure Mode

In the same tests described in the previous section, we also collected the network and CPU utilization at the client stations. Figure 10 compares the client receive data rate for FEC and ODC mode operations. As expected, FEC requires more network bandwidth during normal operation. After the failure, both FEC and ODC use the same amount of bandwidth.

Figure 11 shows the client CPU utilization for FEC and ODC mode operations. At normal-mode, FEC requires more CPU utilization because more data are received and processed. After server failure, they have similar CPU utilization. Note that the CPU utilization for ODC is roughly the same before and after a server failure. This shows that the processing overhead for data reconstruction (after a server failure) is negligible as only simple parity operation is needed.

6. Conclusion

We proposed the Redundant Array of Inexpensive Servers (RAIS) architecture for developing scalable and fault tolerant servers. We compared and contrasted the differences between RAIS and RAID. We studied the challenges in designing a practical RAIS system and proposed our solutions. Our implementation demonstrated that our architecture can achieve fault tolerance at the server level, and support an uninterrupted audio and video services during and after server failure.

References

[1] Y.B. Lee, P.C. Wong, "A Server Array Approach for Video-on-demand Service on Local Area Networks, " *IEEE INFOCOM '96*, San Francisco, USA, March 25-28.

[2] D. Patterson, G. Gibson, R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceeding of the ACM SIGMOD Conference*, June, 1988, pp. 109-116.

[3] Y.B. Lee, P.C. Wong, "Storage Reconstruction in a Video Server Array," *CUHK Technical Report TR-VL-03*, 1995.

[4] Y.B. Lee, P.C. Wong, "Designing a Server Array System for Multimedia World-Wide-Web Services," *CUHK Technical Report TR-WA-01*, 1996.

[5] WebStone 2.0, Silicon Graphics Inc., http://www.sgi.com/Products/WebFORCE/WebStone.

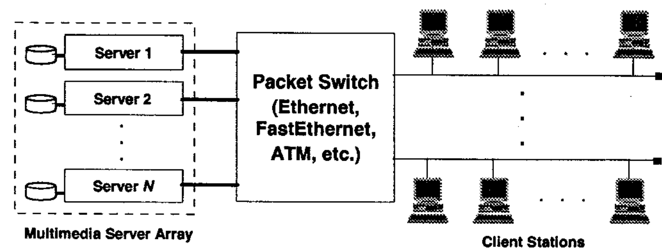

Figure 1 - Server Array Architecture

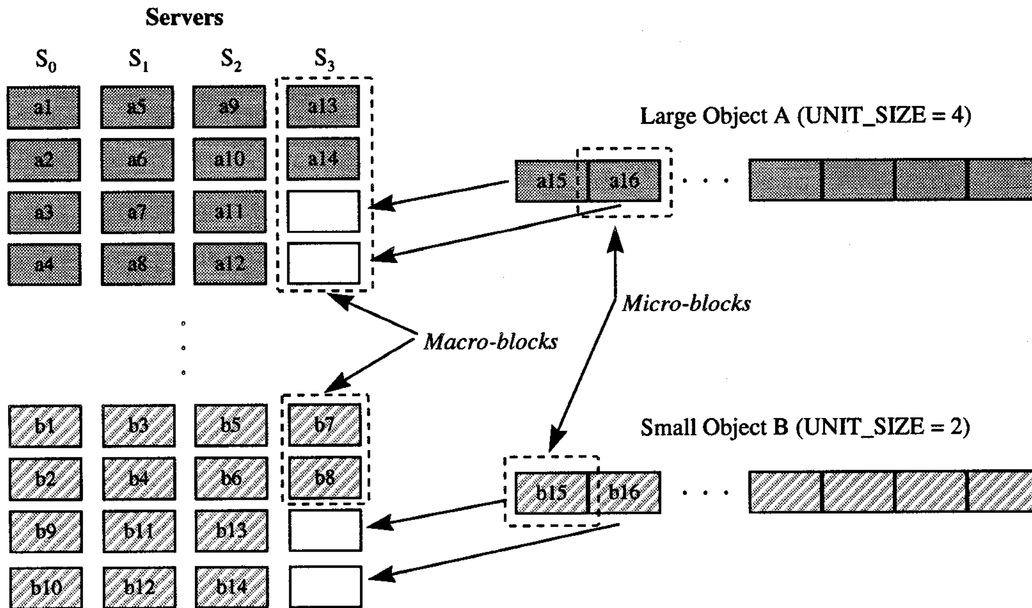

Figure 2 - Dynamic Object Striping

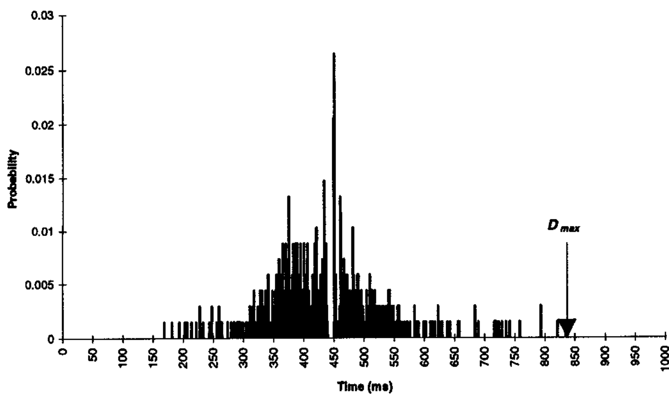

Figure 3 - Distribution for server response time at peak loading

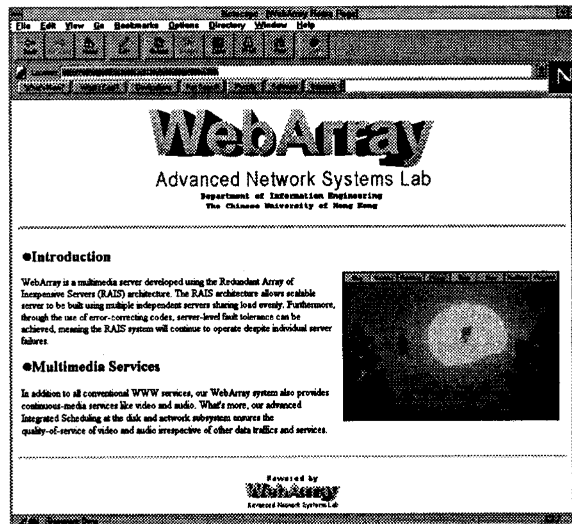

Figure 5 - Accessing WebArray through a WWW browser

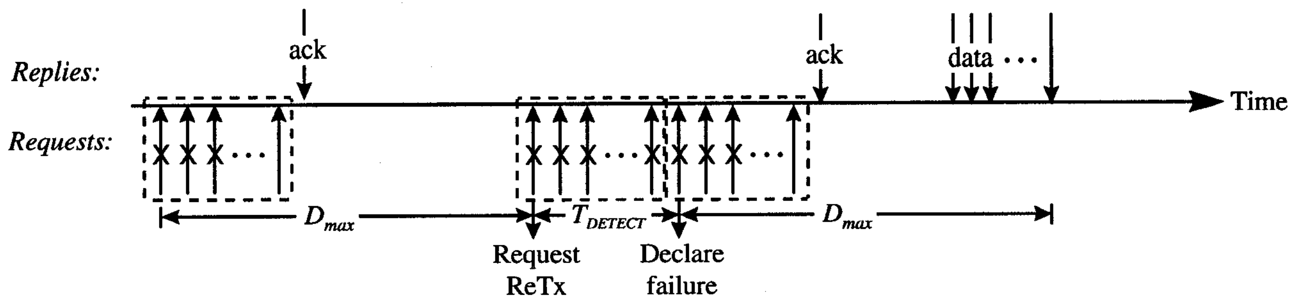

Figure 4 - Worst-case scenario at server failure

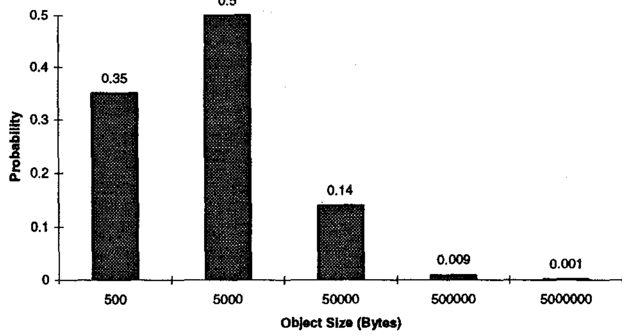

Figure 6 - WebStone standard retrieval distribution.

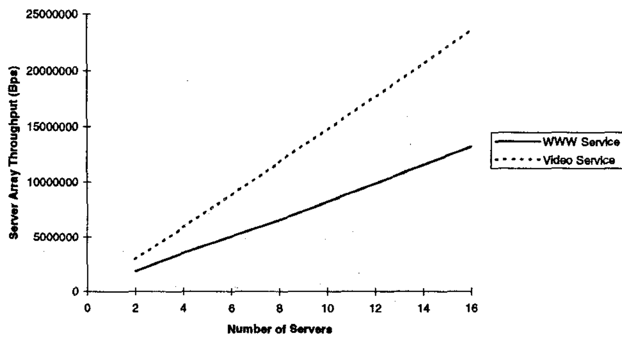

Figure 7 - Aggregate server throughput for WWW and Video services

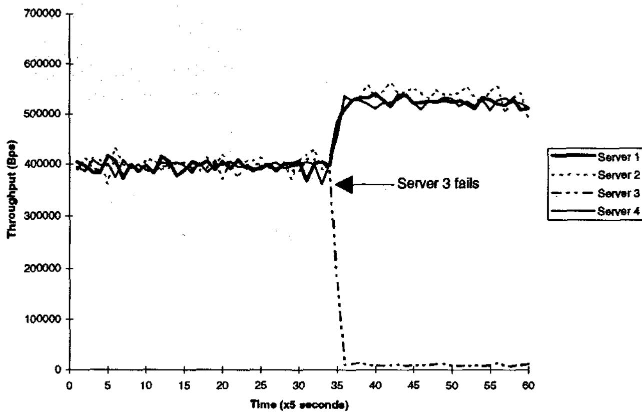

Figure 8 - Test run for single-server failure in a 4-servers RAIS system using ODC fault tolerant

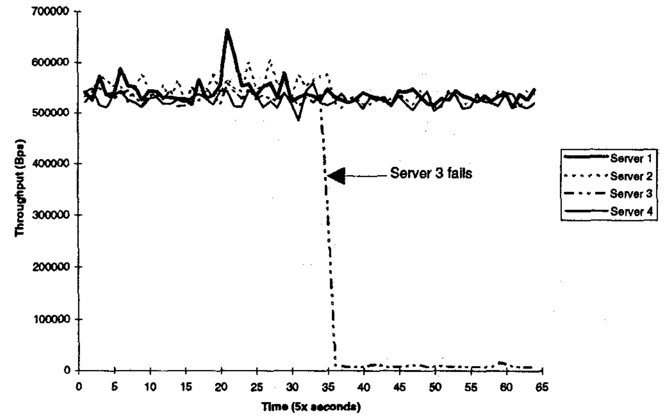

Figure 9 - Test run for single-server failure in a 4-servers RAIS system using FEC fault tolerant

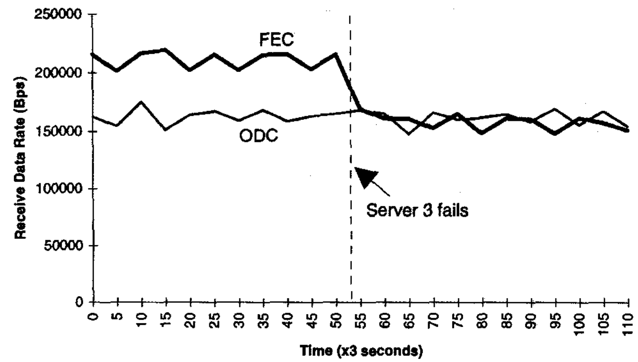

Figure 10 - Client network receive rate before and after server failure

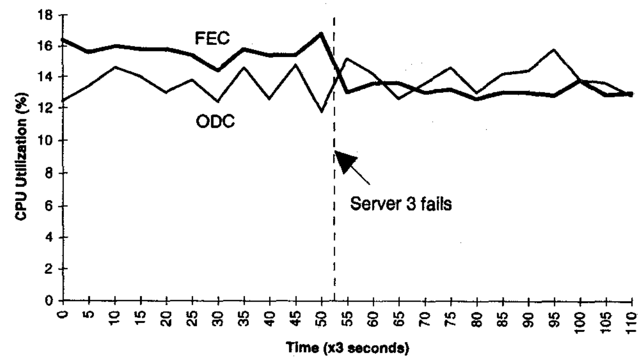

Figure 11 - Client CPU utilization before and after server failure