# Designing a Server Array System for Multimedia World-Wide-Web Services

Y.B. Lee and P.C. Wong
Advanced Network Systems Laboratory
Department of Information Engineering
The Chinese University of Hong Kong, Hong Kong
Tel:(852)-2609-8372 Fax:(852)-2603-5032
{yblee, pcwong}@ie.cuhk.edu.hk

**Abstract**

*In [1] we reported the design and implementation of a video-on-demand (VOD) system using a server array architecture. In this paper, we extend the work into a full multimedia server array system for world-wide-web services. The system allows stream-type services such as video and audio guaranteed continuous service irrespective of the background data traffic. The server array approach has the benefits of (1) more system capacity, as individual server has individual CPU, disk, and network channels, (2) scalable, as the system capacity increases with the number of servers without the need for data duplication, and (3) fault tolerant, as server-level fault-tolerant schemes can be devised for sustaining service even if a server fails. We describe the design and implementation of our WebArray system - a server array software system developed for multimedia world-wide-web services. Our implementation employs integrated scheduling at the disk and network subsystems to ensure the continuous delivery of video and audio.*

## 1. Introduction

The exploding success of world-wide-web (WWW) creates many exciting applications in the Internet. The ease of use, flexibility and portability of web documents makes it an ideal tool for networked information retrieval. The Internet web servers and browsers are already making their way into business corporate information systems for *intranet* applications.

A general web system consists of *one* or *more web servers* connected to the network using TCP/IP. Users retrieve information using *web browsers* like Netscape or Internet Explorer to contact the web servers for information retrieval. The browser will handle all network communications with the server and presents the retrieved data in the appropriate format. The current web servers support many data types, including text, formatted hyper-text (HTML), images (GIF, JPEG), audio (au, wav), animation, or even downloadable components like Java applets or ShockWave Director applets. While there are some streaming audio and video extensions being introduced, none of them provide quality-of-service control and the video and audio streams will suffer significantly if the server is busy serving web documents.

All current web servers run on a single server. Large and busy web sites may run server clusters and distribute requests to individual servers for a particular piece of information - *Service partitioning* [2-4]. This technique allows the loading to be shared by several servers, but if many clients are accessing one particular server for a specific page (e.g., a popular video), the *hot-spot* server will still be overloaded. To remedy this problem, one may replicate the more popular pages on several servers - *Service replication*. This generates into management complexity, and the storage needed will be several times of the original storage. Given that good quality video and audio streams require vast amount of storage (e.g., a 90-minute MPEG 1 movie requires approximately 1 GByte storage), service replication clearly has its limitation.

In this paper, we consider a multimedia server array system where data blocks of each stream (esp. video and audio) is striped across an array of autonomous servers - *Service Striping*. The client contacts the servers one by one to request blocks for its own stream, and reconstruct the stream at the client. In this way, the loading of client requests is uniformly shared by all the servers without the need for data replication. Such an approach has the benefits of (1) more system capacity, as individual server has individual CPU, disk, and network channels, (2) scalable, as more client requests for one service can be supported by adding more servers, and (3) fault tolerant, as server-level fault-tolerant schemes can be devised so that when a server fails, the system can still maintain service.

The rest of the paper is organized as follows. Section 2 describes the general server array architecture. Section 3 describes the server striping and data placement policy issue and presents our proposed Dynamic Object Placement scheme to balance storage and I/O efficiency. Section 4 describes the design of our multimedia server array system. Section 5 describes our implementation and presents some experimental results. Section 6 gives our conclusion and outlines some future work.

## 2. Server Array

A server is an entity serving requests posted by the clients. The server retrieves the data blocks from the storage, and sends back the blocks in the form of packets via the network. Under this view of a server, the server capacity can be limited by three factors: storage device (e.g. disk) throughput, CPU processing and I/O capability, and network access bandwidth. To increase the system capacity, one may use (1) a faster storage device, e.g. disk array instead of a single disk, (2) a more powerful CPU, or (3) a high-speed network such as Fast Ethernet or ATM. However, this per-component upgrade approach does not provide a long-term scaling path for an even higher server capacity, and is not always possible.

Consider a disk array used to increase the disk throughput. The effectiveness of such approach is limited by the reduction in per-disk block size. For example, if the block size for retrieval is $Q$ bytes, then the per-disk block size for an $N$-disk array will be $Q/N$. For large disk arrays, this reduction in block size can hamper the throughput of the disk array as disk seek time and latency overhead will be significant.

A server array, on the other hand, consists of an array of autonomous servers. 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 at server #1, block #2 at server #2, and so on. A fast packet switch fabric is used to connect servers and client stations, and each client will contact the server one by one for retrieving blocks of a particular stream, and reconstruct back the stream at the client for its use. Since each server has its own storage subsystem, CPU, and network channel, the overall server capacity increases with the number of servers, and one may increase the capacity at any time by adding a further server and redistributing the data over all the servers. We note an asymmetric traffic requirement between servers and clients. So the servers may be assigned with one or more high-speed links, whereas several clients may share a single low-speed link.

## 3. Data Striping and Placement Policy

The key to achieve load-sharing without data duplication in server array is the distribution of data across all servers - *data striping*. This section considers how to distribute data blocks over a server array to maximize the retrieval efficiency.

While data striping has been studied extensively in the disk array context, data striping in server arrays poses new challenges. Servers are loosely coupled using a communications network which is subject to all kinds of delays, loss, and bandwidth limitation. So server array cannot use small striping unit size like disk array, otherwise retrieval will be very inefficient. On the other hand, data striping in server arrays is done at the application level. Unlike disk array which requires all data to be striped using a uniform scheme and unit size, server array allows files or services to be striped differently, providing different levels of fault tolerance. Finally, server array can achieve server-level fault-tolerance by employing data redundancy similar to that of the Redundant-Array-of-Inexpensive-Disks (RAID). Here, we considered the notion of a server-level fault-tolerant server array system - Redundant-Array-of-Inexpensive-Servers (RAIS). The use of a server-level fault-tolerant scheme can cover disk failures as well. For example, a disk failure in one particular server simply manifests as a server failure. Once detected, the system continues to function until the failed disk is repaired.

It is interesting to note that while RAID schemes are useful in data applications for protecting from disk failures, they have a serious problem when used for multimedia applications. This is because in multimedia systems, most traffic are continuous-media services. In the event of disk failure, each subsequent lost block has to be reconstructed by reading blocks of the same stripe (even if they have been read previously as stream-type services are mostly sequentially accessed) from the remaining active servers. The effective throughput of the RAID can be significantly lower than normal. This is unacceptable as continuous services like video and audio requires stringent server response time.

### 3.1. Storage Striping

In a server array, there are two possible ways of data striping: storage striping and object striping. In storage striping, storage in each server is divided into equal-sized blocks of $Q$ bytes. For ease of description, we denote the servers in an $N$-servers array as $S_0, S_1, ..., S_{N-1}$. Within each server, we divide all storage spaces into fixed-size blocks $b_{j,i}$, denoting the $j^{th}$ block in server $i$ (Figure 2).

All blocks from the same row comprise a *storage stripe*. Under fault-tolerance, parity blocks might be

allocated to protect the system from server failures. Any lost block in a stripe can then be obtained by a simple exclusive-or operation on the remaining data blocks in the same stripe. Multiple-server failures can also be protected by using more redundancy through Reed-Solomon encoding. In such cases, a ($N$+$k$)-servers array will be able to protect up to $k$ simultaneous server failures with redundancy overhead of $k$/$N$.

The size of striping units determines the efficiency of data retrieval from the storage subsystem. To increase effective disk throughput, one should use large block size. However, large block size is inefficient for storing small data objects, such as text pages in a multimedia system environment. Consider that we need to retrieve $N$ blocks out of the ($N$+$k$) blocks in a stripe when a server fails. If the data object is smaller than a single stripe, we still need to read the entire stripe for reconstruction. We consider the following two problems in using large striping units:

1. *Internal fragmentation* - data objects may be smaller than a storage stripe unit, thus wasting the storage space within a stripe unit;
2. *Striping overhead* - data objects may be smaller than a stripe, thus requires reading extra storage stripe units for reconstruction during failure.

The second problem is easily solved by using a truncated-stripe to cover the object if it is smaller than a complete stripe. For example, if the server array consists of $N$ servers and the data object only needs $m$ ($m < N$) units (including the parity block), we will just use $m$ units to store the object. During server failure, we will need to transfer any $m$-$1$ out of the $m$ stripe units to the client stations for reconstruction. Note that this setting requires that the server supports variable stripe lengths for different objects. This can easily be implemented as part of the directory service within the system.

Still, the first problem cannot be solved with storage striping as all objects are striped with a uniform striping unit size. On the other hand, we observe that it may be desirable for different services to be striped in a different manner, i.e., with different stripe unit size and redundancy levels. We consider in the following an alternative to storage striping - *object striping*.

### 3.2. Object Striping

Object striping does striping at the data object level. In other words, the striping is performed on an individual data object, such as a HTML page, an image, audio, or video file. Note that an object in this context is not equivalent to a disk file. We could have compound objects which comprise multiple different types of objects.

We consider a Dynamic Object Placement (DOP) scheme to optimize striping for various kinds of media data. In DOP, all storage and objects are striped using a small stripe unit (e.g. 1KB), called *micro-blocks*. Using small storage stripe units solves the internal fragmentation problem. However, the placement policy of the object stripe units is not fixed, and depends on the size of the data object to optimize efficiency.

Under DOP, each stored data object has a *DOP policy* consisting of three attributes:

{START_UNIT, UNIT_SIZE, REDUNDANCY}

START_UNIT records the starting stripe unit of the particular data object; UNIT_SIZE records the size of a *macro-block* in units of micro-blocks. The data object will be striped across servers in macro-blocks. Lastly, REDUNDANCY records the level of redundancy employed in storing the data. A redundancy value of zero means no redundancy while a value of $k$ means ($N$+$k$) RS-coding is employed. Note that the conventional server replication technique is a special case of DOP with UNIT_SIZE equal to size of data object, and REDUNDANCY equal to the number of servers minus 1.

The DOP policy is created when a data object is first stored into the server array. The DOP policy may be changed and consequently the data rearranged if necessary.

The DOP policy allows different types of data objects to be striped in a different manner. Figure 3 shows that large objects can be assigned DOP policy with large UNIT_SIZE (e.g. 64 microblocks for video streams) for optimizing I/O efficiency at the disk subsystem. So each client request will retrieve 64 microblocks from one server at a time. Smaller objects or infrequently accessed data objects can be assigned DOP policy with small UNIT_SIZE (e.g. one or two microblocks). On the other hand, different objects may have different redundancy levels as well. For example, video objects can be assigned with REDUNDANCY of 1, web page graphics with REDUNDANCY of 2, and web page texts with REDUNDANCY of 3. In this way, the video service survives single-server failures, while graphics and web page texts will survive double and triple-server failures respectively. This flexibility allows the multimedia service to degrade gracefully during multiple server failures.

## 4. Multimedia System Design

In a multimedia server array, each server is by itself an autonomous multimedia server interacting with the client stations. In this Section, we consider a software and protocol architecture as shown in Figure 4 for supporting multimedia services in an integrated manner.

At the top layer is the Integrated Services Server (ISS), acting as the service manager for various services and system components. It is responsible for initializing all system components, including the disk and network

drivers, disk and network schedulers (IDS, ITS), as well as the individual services (e.g., web pages, audio, video). Note that these services are not the same as their conventional counterparts. They participate in the server array to serve a subset (rather than the whole) of striping units upon each request from the clients.

To illustrate the system operation, when a request arrive at the multimedia server array, the location and placement of data objects are resolved by the Server Array Directory Service (SADS). Then the data objects are retrieved through an Integrated Disk Scheduler from the disk storage into memory buffers. The retrieved data will then be scheduled for transmission by the multimedia transport protocols and the Integrated Transmission Scheduler. We will discuss each component in the following.

### 4.1. Server Array Directory Services

Data objects in a server array are likely to be distributed across multiple servers and various levels of redundancies are supported. To provide location transparency to the client applications, the Server Array Directory Services (SADS) acts as an agent to manage and resolve object names to data location mappings. This allows client applications to retrieve objects by name regardless of where the data objects is stored within the server array.

### 4.2. Integrated Disk Scheduler

After a client request is resolved by the SADS, the data object can then be retrieved from the disk storage. Note that the retrieval of data object must meet the QOS requirements for the supported services. For example, video requests have to be handled in such a way that the retrieval time is short enough to prevent client video playback starvation. This is complicated by the existence of other data requests, for example, HTTP text retrieval or FTP service.

The problem is how to resolve disk access contention among various services. To this end, we observe two potential problems in the disk retrieval process:

1. *Requests queueing* - time-critical requests like video can be delayed by non-time-critical requests arrived earlier.
2. *Requests blocking* - time-critical requests can be blocked for an extended period by a single large request inside service.

To solve the first problem, we use an Integrated Disk Scheduler (IDS) with multiple static-priority queues to assign a higher priority of access to time-critical requests. To solve the second problem, the IDS sets a limit on the maximum allowed data block size on each request serviced. Requests for larger block sizes are divided into smaller sub-requests of maximum size $Q$ and then submitted to the disk scheduler one by one.

Consider that there are $L$ queues, denoted as $Q_0$, $Q_1$, ..., $Q_{L-1}$, in the IDS. Let each queue $i$ be assigned a unique fixed priority $p_i$, with a lower value represents a higher priority. Each queue also has a maximum concurrency limit $C_i$, which governs at most how many requests of that priority can be serviced simultaneously by the disk. Finally, there is a single maximum concurrency limit $C_{disk}$ for the disk device itself to limit the number of requests (from any priority) to be served concurrently. This is necessary as the requests are submitted asynchronously to the disk device so that it can optimize efficiently through disk seek scheduling (e.g. SCAN).

The number of queues and their priorities are assigned at system start-up by the system manager using the Integrated Services Server. For example, the system manager may prefer 60% of the disk throughput be reserved for video and audio. The number of data requests to be serviced at any time must be limited to 40% by setting the appropriate values for $C_i$ of data queues. We can register each service on one or more request queues where future disk I/O requests will be directed to.

Figure 5 gives the scheduling algorithm for the IDS to serve the request queues. The scheduling algorithm ensures that lower priority requests are not served unless all higher-priority queues either have no queueing requests or have already reached their concurrency limit. In our implementation of a multimedia server array, we have three priority queues: $Q_0$ (highest priority) is assigned for continuous-media service, *Q1* is assigned for HTTP, and $Q_2$ (lowest priority) assigned for FTP service. HTTP service is assigned a higher priority as web browsing is often interactive while FTP may take place in the background.

### 4.3. Multimedia Transport Protocols

In this section, we describe the multimedia transport protocols used to deliver the retrieved data through the network to the client stations.

A multimedia server array consists of multiple application-level protocols for different types of services. Examples include HTTP service for World-Wide-Web applications, FTP service for remote file transfer, and STP service for on-demand continuous media like video and audio. Unlike their conventional counter-parts, which assumes *single-server-multiple-client* operations, the STP, HTTP, and FTP protocols in our server array are capable of handling data objects distributed across an array of autonomous servers. In addition, server-level fault-tolerance capabilities are available through data redundancy and support by the lower-layer network protocols.

To support the above capabilities for application layer protocols, we designed two protocol suites for data and continuous stream services having the following features:

1. *Fault-detection capability* - the application protocol must be able to detect server failures efficiently and take appropriate actions to mask the failure (see below);
2. *Fault-tolerance transparency* - retrieval of an object during server failure should be transparent to the user unless the failure exceeds the fault-tolerance capability for the object;
3. *Quality-of-service* - the QOS requirement (if any) of a service must be maintained.

While there are many studies on multimedia protocols [5-7], their emphasis are on the third point, i.e. QOS requirement for individual services. In [1] we proposed a protocol for a video server array, meeting the placement transparency and QOS requirements. In this study, we extend the work to full multimedia services in a multimedia server array with support for server-level fault-tolerance.

Rather than describing each protocol shown in Figure 4 in detail, we will briefly describe their functions and focus on the solutions they provide to meet the capabilities posed earlier. The well-known HTTP and FTP protocols are standard Internet protocols and will not be discussed here.

**Data Transport Protocol.** The Data Transport Protocol (DTP) provides connection-oriented data delivery services similar to the Transmission-Control-Protocol (TCP) used in the Internet. However, in order to support server array and fault-tolerance, extra mechanisms are required. Firstly, each connection from client to the server array is a one-to-many connection instead of the conventional one-to-one connection. Secondly, as a data object could be retrieved from multiple servers, data sequencing and reconstruction has to be done before passing to the upper layers. Thirdly, failure-detection mechanism has been added to detect and indicate server failures to upper layers for failure recovery.

**Unreliable Datagram Protocol.** The Unreliable Datagram Protocol is a thin layer on top of the underlying datagram service (e.g., UDP). The primary functions of this layer is to hide the lower layer protocol complexities and provide buffered asynchronous transmit and receive services.

**Reliable Datagram Protocol.** The Reliable Datagram Protocol (RDP) is crucial to the fault-tolerance capability. Specifically, detection of server failures is performed by the RDP and once detected, upper-layer protocols are signalled to recover the failure. While failure-detection also exists in the Data Transport Protocol, the algorithm employed at the RDP layer is designed for real-time failure detection and recovery for supporting stream-type services, so that these services will maintain continuity even if a server fails.

To establish a connection using RDP, the service requester (e.g. STP) has to define failure condition for the requested connection. Failure conditions can be defined by the tuple:

$$\{ \text{MAX\_RETX}, \text{TIMEOUT} \}$$

MAX_RETX is the maximum number of retransmissions to attempt before declaring the connection failed. TIMEOUT is simply the maximum time to wait for an acknowledgement before retransmitting the unacknowledged datagram.

Hence the maximum time for detecting failure is simply given by:

$$T_d = \text{MAX\_RETX} * \text{TIMEOUT}$$

Therefore a system designer can adjust the failure condition tuple according to the actual system characteristics. The failure condition cannot be set too loose or failure may be detected too late and media playback affected. On the other hand, it cannot be set too strict or false alarms will be generated. The maximum failure detection time can be used to determine the buffer requirement for continuous media playback during failure as discussed in [1].

**Stream Transport Protocol.** The Stream Transport Protocol is a revised version of the Video Transport Protocol as discussed in [1]. The main idea is to ensure that video can be delivered to the client in a timely manner, and the client buffer will never be starved of data for supplying video blocks to the decoder for playback. Consequently, video can maintain continuity at the client stations. Interested readers may refer to [1] for details.

### 4.4. Integrated Transmission Scheduler

Similar contention problem for disk access happens at the network access. For example, the HTTP server may transmit a large block of data (e.g. a JPEG file) while a video block is being transmitted by the network driver. In many cases, the order and pattern of transmission of both blocks simultaneously are implementation dependent. This is clearly undesirable as video traffic has far more stringent timing constraint than data traffic.

To solve the above problem, we need a mechanism to control which transmission requests are served by the underlying network drivers. The result is the Integrated Transmission Scheduler which sits between the upper

network protocols and the lower network drivers as depicted in Figure 6.

There are two components within the ITS, the first part is a collection of static priority queues, each holding requests from their respective priority. All queued requests are scheduled for transmission in the second part of the ITS using the same scheduling algorithm as employed at the Integrated Disk Scheduler. The scheduling algorithm ensures that higher priority requests (e.g. video, audio) will not be blocked by lower-priority requests (e.g. HTTP, FTP) at the transmission queues in the Traffic Shaper.

The Traffic Shaper in the ITS comprises $Z$ transmission queues, each holding at most one block ($Q$ bytes) of data of a request for transmission. However, transmission is not performed on a block-by-block basis, rather a sub-block or packet ($G$ bytes) is transmitted from each transmission queue in each round of transmission, in an interleaved manner. Hence, transmission of up to $Z$ data blocks can proceed simultaneously by multiplexing transmission at the network driver.

This approach is useful for shaping the out-going traffic to avoid transmitting data at large bursts, which would be the case for FIFO discipline transmissions. Experimental results [1] have shown that such traffic shaping can reduce packet loss significantly at the client stations. This is important as servers are normally high performance computers with high speed networks, whereas clients are low-end computers which might be busy in other tasks. The traffic shaper can effectively bridge the bandwidth gap between the servers and the clients.

## 5. Implementation Results

To demonstrate the concepts and designs, we developed a multimedia web server at the Chinese University of Hong Kong. At the time of writing, we have completed the server-array version of the continuous-media server (STP) and HTTP server, and operate our system with four autonomous servers. The two services are controlled and scheduled by the IDS and ITS. In the following sections, we will presents our results so far and analyze their implications. Other services like FTP will be implemented in the near future.

### 5.1. Test-bed Configuration

All results are obtained from our multimedia server array implementation - *WebArray* on four Pentium-90 PCs running the Windows NT operating system. The detail system configuration is listed in Table 1.

HTTP requests are generated using WebStone 2.0 [8] and the standard test-file sizes distribution are shown in Figure 7. For STP service, requests are generated using client PCs. Each client PC establishes an active STP connection for viewing a 1.2Mbps MPEG 1 video, which is a 30 fps, near TV quality video through a web page.

### 5.2. Server Array Capacity and Scalability

In general, the server array architecture is a class of *symmetric* multiprocessing systems where each processing unit (henceforth referred as *processor*) is identical and provides the same service. The scalability of such system depends on several factors:

1. Resource contention among processors
2. Communications among processors
3. Synchronization among processors
4. Processing overhead due to parallelization

For a server array, we see that there are no resource contentions among servers as each server has individual disks, CPU, and network connections. Secondly, communications among servers are in general not required as requests are all generated by client stations and sent to the specific server directly. Thirdly, there is no synchronization among the servers as each server operates autonomously without information on other servers in the server array. Hence the first three factors do not exist in the multimedia server array architecture. We study below the fourth factor on scalability which is dependent on different kinds of services.

### 5.3. Analyzing the HTTP Service

In HTTP 1.0 [9], there are several supported requests, including GET, HEAD, and POST. Among them, GET is used to request data objects from a HTTP server. As information retrieval forms the highest traffic loading to a HTTP server, we only consider the GET request in the following.

The syntax of the GET request is as follow:

```
"GET URL HTTP/1.0"
```

where URL (*uniform-resource-locator)* identifies a data object at the server. For example,

```
"GET /file25k.html HTTP/1.0"
```

requests a file object named `file25k.html` from the server. To simplify analysis, we will assume there are $m$ objects in the server, denoted as the set $\{\boldsymbol{F}\}$, and the size of object $i$ denoted as $f_i$ is $u_i$. The access probability of the object set is represented by $\{\boldsymbol{P}\}$ where $p_i$ is the probability of file $i$ being requested for each request generation at the client stations. Requests are assumed to be generated independently.

In a server array, where data objects are no longer stored in a single server, a GET request has to be converted into multiple GET requests, each requesting one part of the data object from one server. The HTTP client

service at the client station will re-assemble the stripe units and pass to the upper layer (e.g. a web browser).

Under this model there are three possible area of inefficiencies: (a) increased number of connections per client per server; (b) increased number of transactions per client on each server; and (c) decreased I/O efficiency.

If we assume *full striping* for all data objects, i.e. for any data object of size $K$ bytes, we store $K/N$ bytes into each server, then the net effect is the same as converting the set of file objects $\{\boldsymbol{F}\}$ into a new set of file objects $\{\boldsymbol{F'}\}$ in each server, where object $i$ has size $u_i/N$. As one might expect, the exact decrease in server performance depends on the implementation as well as the hardware and software platform. Figure 8 compares the server throughput at different object sizes to demonstrate the I/O efficiency problem. For real-world data objects, we benchmarked server arrays of 1, 2, and 4 servers with 20 HTTP clients using the standard WebStone 2.0 data set and access distribution and the results are shown in Figure 9. The results show that the reduction in object size due to full striping impose considerable overhead in processing and hamper the I/O efficiency. Hence the per-server capacity decreased for more servers. This problem could seriously restrict the size of the server array system.

Fortunately, the Dynamic Object Placement scheme proposed in Section 3 provides a solution. Specifically, we could set a lower bound on the UNIT_SIZE to improve I/O efficiency for smaller objects. For example, we benchmark the same 4-servers WebArray system using minimum UNIT_SIZE of 5KB, 50KB, and 250KB respectively and the server capacity is significantly improved (Figure 10). Note that an UNIT_SIZE of zero represents full striping (i.e. no lower limit on UNIT_SIZE) and duplication represents full data duplication at all servers (i.e. no striping). According to Figure 10, we can closely match the original server throughput by using minimum UNIT_SIZE as small as 5KB. Hence by using DOP with limited minimum UNIT_SIZE, the WebArray system can be scaled-up without significant loss in I/O efficiency.

The downside of setting a minimum UNIT_SIZE striping size is that some data objects may not be striped across all servers as they are smaller than a full storage stripe, in which case redundancy (if employed) will become significant as the stripe size is decreased. However, this should not pose much problem as the object in question is small.

### 5.4. Analyzing the STP Service

For the STP service, there will be no decrease in I/O efficiency as the stream-type objects like video and audio are much larger than the striping unit size. The STP protocol is designed specifically to minimize processing overhead in a server array system and hence does not have significant extra processing requirements.

We obtain results for our WebArray with 1, 2, and 4 servers on a switched 10 Mbps Ethernet. We monitor the server CPU utilization as an indication of server loading. Figure 11 shows that a single P5-90 server can support up to 10 STP sessions (~12Mbps), and the server CPU utilization is roughly proportional to the number of active clients. On the other hand, the CPU utilization is reduced in a linear fashion when more servers are added. The result is for P5-90 servers with SCSI disk and 10 Mbps Ethernet adapters. Preliminary tests show that a P5-166 with a 100 Mbps Fast Ethernet and a Fast-and-wide SCSI disk achieve more than doubled the server capacity.

### 5.5. Integrated Disk and Transmission Scheduling

In this section we analyze the performance of the Integrated Disk Scheduler and the Integrated Transmission Scheduler in scheduling continuous-media and data services. Our current WebArray implementation supports both HTTP and STP services. We run benchmarks to measure the throughput of both services by varying the number of STP sessions while concurrently serving 20 HTTP clients simulated using WebStone 2.0 with the standard file set and access distribution. The results are summarized in Figure 12.

We observe that the HTTP service throughput drops progressively as more STP sessions are established. On the other hand, the STP service throughput increase proportionally to the number of STP sessions. This demonstrates the effectiveness of the Integrated Disk Scheduler and the Integrated Transmission Scheduler in giving priority to STP service. If we run both services without either one of the schedulers, the STP service will be disrupted, resulting in video playback interruptions.

In the same test we also collected the mean system time (service + queueing) for both services and shown in Figure 13. While the HTTP system time increases with increasing STP loading, the STP system time remains below 250 ms regardless of the server loading. This again suggests that the I/O schedulers indeed protect the STP service from the HTTP loading at the expense of increased HTTP system time.

To ensure that HTTP sessions are not totally denied service, it is crucial to limit the number of STP sessions on a server by some admission control algorithms. This ensures that the system will have a sufficiently large spare capacity to provide a reasonably good service (i.e., delay) to HTTP services.

## 6. Conclusion

The proposed multimedia server array architecture in this paper provides a practical design framework for implementing truly scalable, and fault-tolerance multimedia servers. While the general issue of disk and network resource allocation and scheduling has been solved by the I/O schedulers, there are other issues yet to be tackled, including (a) CPU scheduling; (b) scheduling of CGI programs in the HTTP server; and (c) integration with other server extensions for dynamically generated data (e.g. back-end database server). Yet, we show that integrated scheduling is essential at both the disk and network to achieve service guarantee for stream-type service irrespective of the background data service (e.g. HTTP, FTP). Our current implementation supports both server-array version of HTTP and STP services with load-sharing among servers, scalable server capacity, and server-level fault-tolerance.

## Acknowledgements

The research and implementation are supported by the Hong Kong Government Industrial Department.

## 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] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the SUN Network File System," *In Proceedings of the Summer Usenix Conference*, 1985.

[3] J. H. Howard, M. J. Kazar, S. G. Menees, D. A. Nichols, M. Satyananrayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6(1), 1988.

[4] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, D. C. Steere, "Coda: A Highly Available File System for a Distrbuted Workstation," *IEEE Transactions on Computers*, vol. 39(4), pp. 447-59, 1990.

[5] F.A. Tobagi, J. Pang, "StarWorks™ - A Video Applications Server," *IEEE COMPCON Spring '93*.

[6] P.V. Rangan, H.M. Vin, S. Ramanathan, "Designing an On-Demand Multimedia Service," *IEEE Communications Magazine*, Vol.30, No.7, July 1992, pp. 56-65.

[7] J. Y. Hui, E. Karasan, J. Li, and J. Zhang, "Client-Server Synchronization and Buffering for Variable Rate Multimedia Retrievals," *IEEE Journal on Selected Areas in Communications*, vol 14(1), January 1996.

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

[9] Hypertext Transfer Protocol - HTTP/1.0, *RFC1945*, http://ds.internic.net/rfc/rfc1945.txt.

| Hardware/Software | Platforms |
|---|---|
| Clients | Intel PCI-based 486DX-66 or P5 PCs |
| Servers | Intel PCI-based Pentium-90 PCs with 32MB RAM |
| Server Storage | 8 GB (An array of four SCSI-II 2GB harddisks) |
| Network | PCI-based 100 Mbps FastEthernet |
| Switch | 16x16 Bay Networks 28115 FastEthernet Switch |
| Web Browser | Netscape Navigator, Microsoft Internet Explorer, and WebStone (benchmarking) |
| Video Decoder | Sigma Designs RealMagic MPEG decoder |
| Client Operating System | Microsoft Windows for Workgroup with Microsoft TCP/IP-32 or Windows 95 |
| Server Operating System | Microsoft Windows NT 3.51 |

**Table 1. Testbed Configurations.**

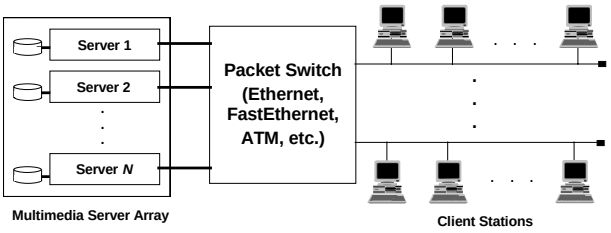


**Figure 1. Server Array Architecture.**

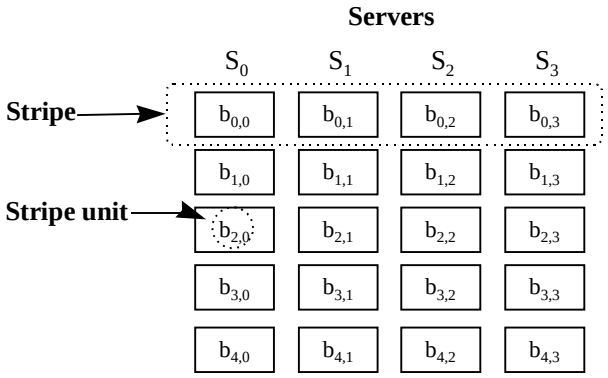


**Figure 2. Storage Striping.**

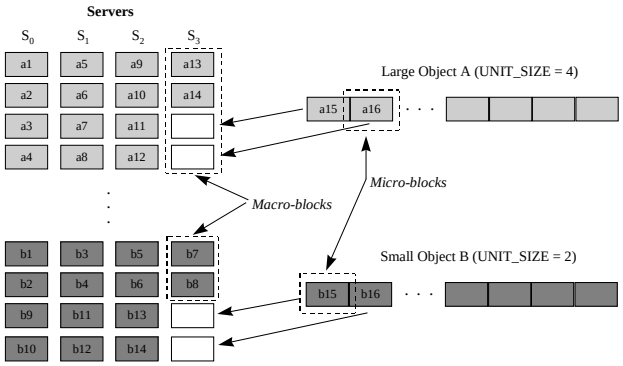


**Figure 3. Dynamic Object Placement Scheme.**

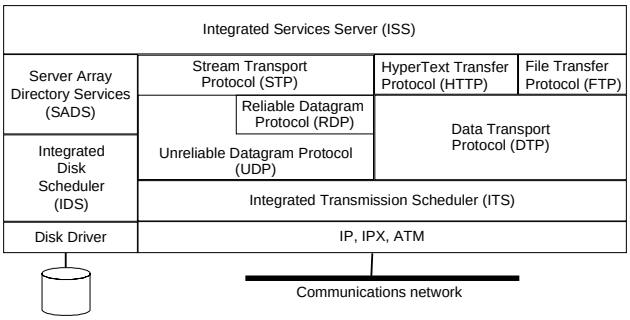


**Figure 4. Multimedia Server Array System.**

```
Algorithm: Integrated Disk Scheduler
While (not terminate)
{
  for (i=0; i<L; i++)
  {
    if (C_disk is reached)
    {
      wait for one request to complete.
    }
    if (Q_i is not empty and
        C_i is not exceeded)
    {
      Remove one request from Q_i
        and submit to disk driver.
      Reset i=0.
    }
  }
}
```

**Figure 5. The Integrated Disk Scheduler Algorithm.**

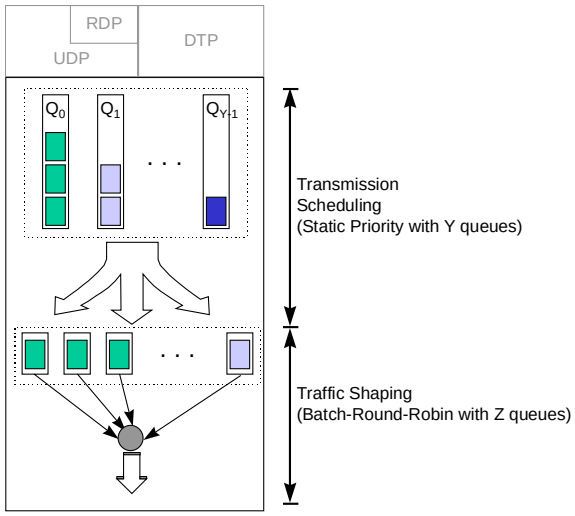


**Figure 6. The Integrated Transmission Scheduler.**

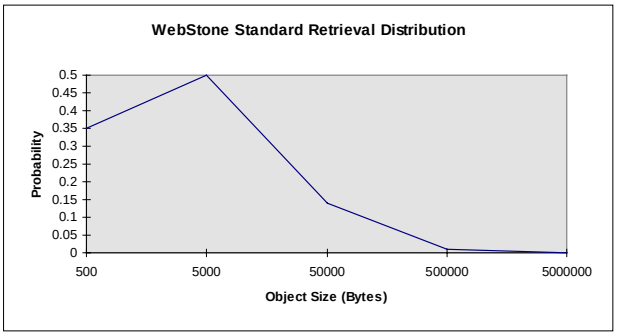


**Figure 7. WebStone retrieval distribution.**

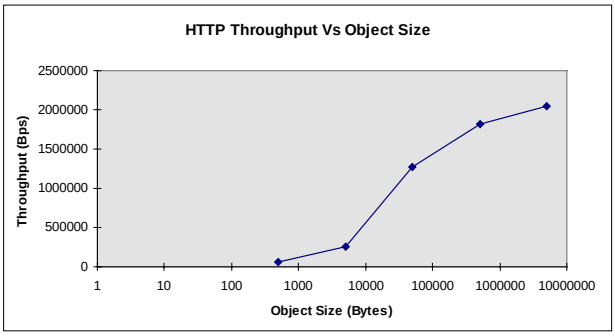


**Figure 8. HTTP Throughput versus Object Size.**

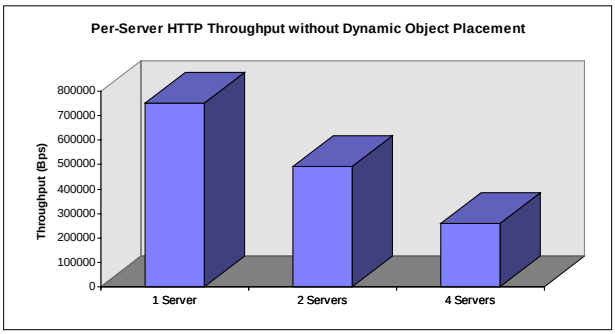


**Figure 9. Per-server HTTP throughput for 1, 2, and 4-servers server array without Dynamic Object Placement.**

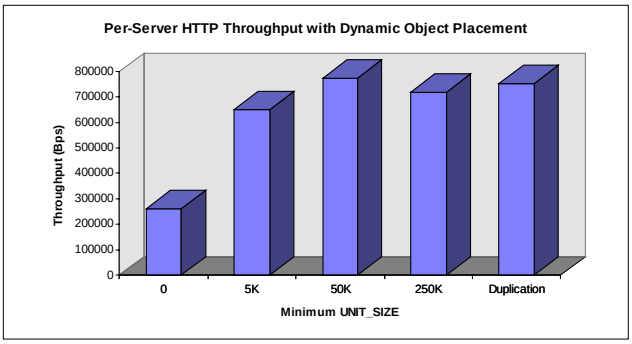


**Figure 10. Per-server throughput for 4-servers server array with Dynamic Object Placement.**

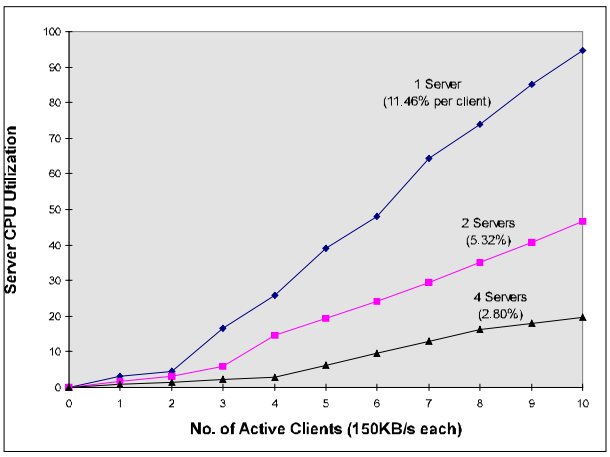


**Figure 11. Server CPU loading for 1, 2, and 4-servers server array versus number of STP sessions.**

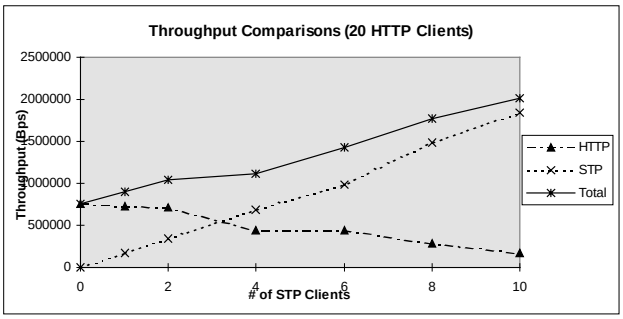


**Figure 12. STP versus HTTP throughput for varying STP loading.**

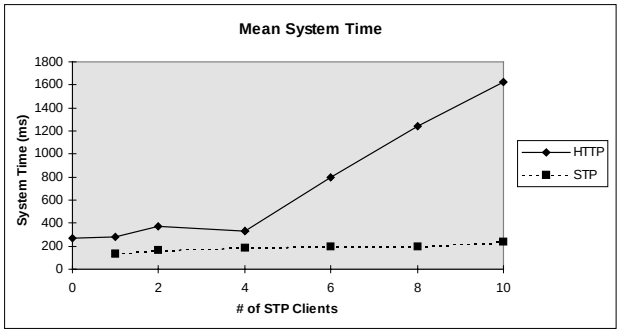


**Figure 13. Mean system time for HTTP and STP for 20 HTTP clients and varying STP loading.**