CS 414 – Multimedia Systems Design
Lecture 30 – Media Server (Part 5)

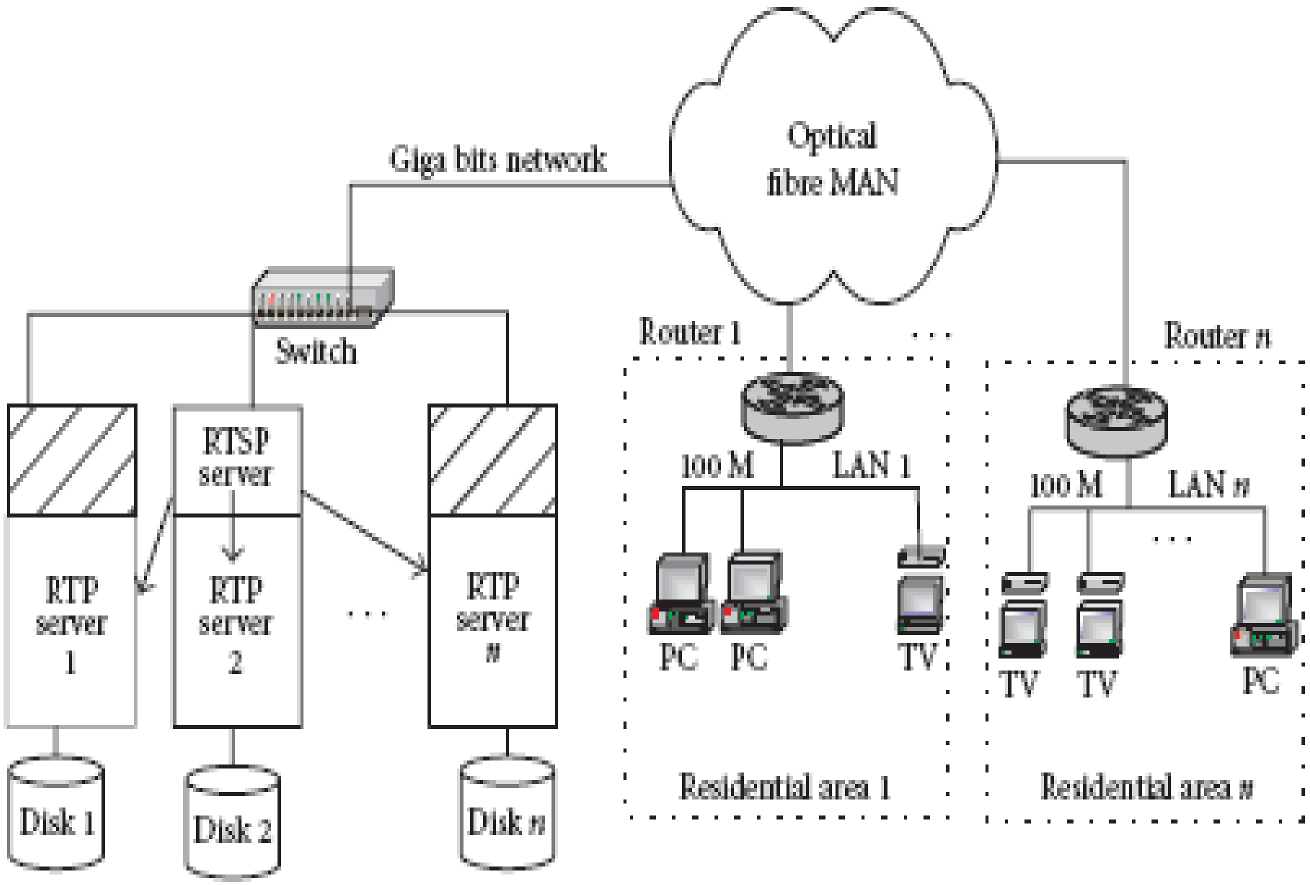
Klara Nahrstedt

Spring 2009

Outline

- Example Media Server Architecture - **Medusa**
  - Media Server Model
  - Validation of Model
- Example of Multimedia File System - **Symphony**
  - Two-level Architecture
  - **Cello** Scheduling Framework at Disk Management level
  - Buffer Subsystem
  - Video Module
  - Caching

Example of Media Server Architecture



Source: Medusa (Parallel Video Servers), Hai Jin, 2004

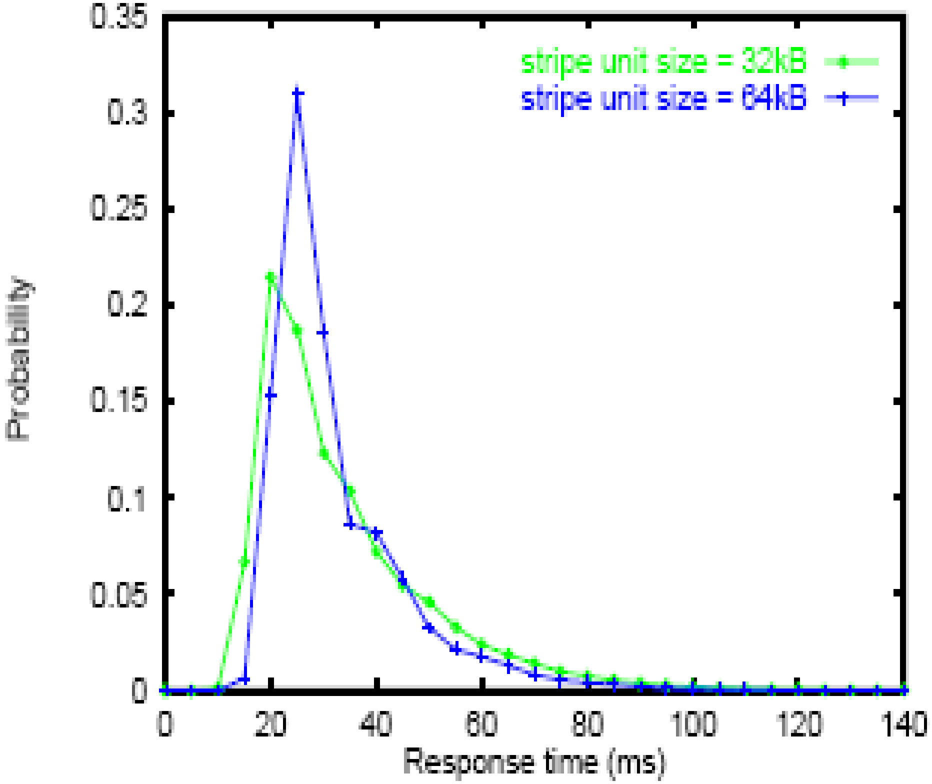
Factors affecting Optimal Block Size

- Server Configuration
  - Number of disks in the array
  - Physical characteristics of disks
  - Type of disk array (i.e., redundant vs. non-redundant
- Client Characteristics
  - Number of clients accessing the server
  - Client request size
- Objective: given server configuration, client characterization, determine the optimal block size

Selecting Metrics

Text

- Aperiodic accesses => client-pull architecture
- Best effort => minimize response time

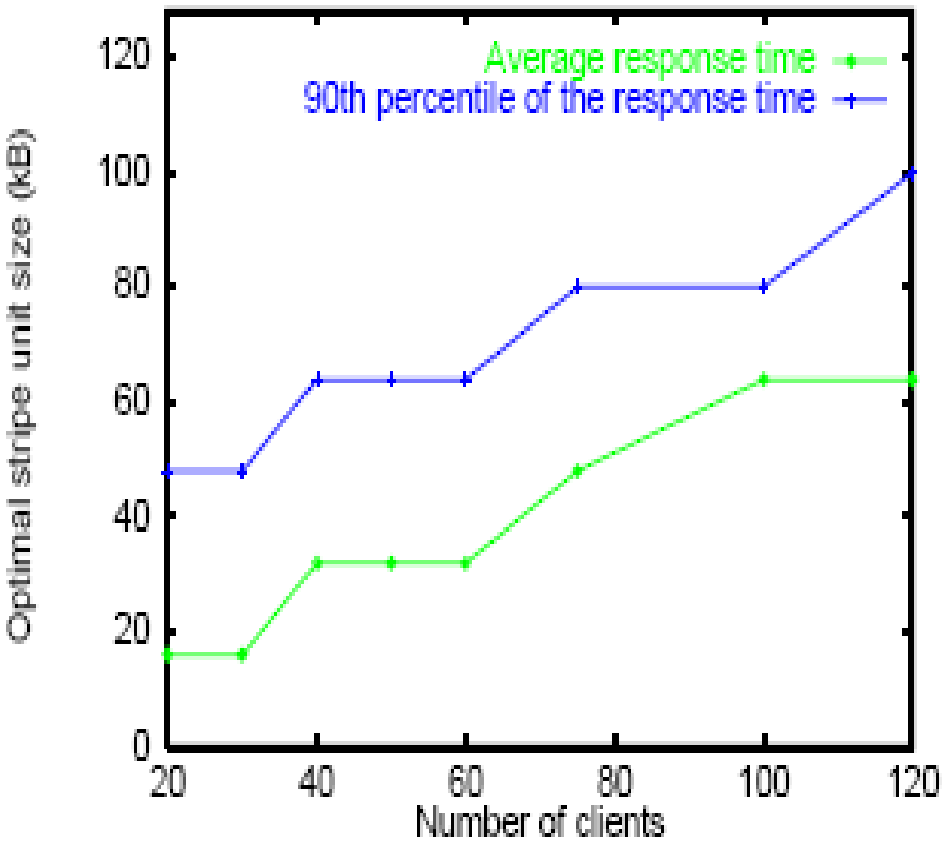


Graph Source: Shenoy Prashant, U. Mass., 2001

Selecting Metric

Continuous Media

- Periodic and sequential access => server-push architecture
- Real-time => minimize tail of response time distribution



Metric: Minimize service time of the most heavily loaded disk

Graph Source: Shenoy Prashant, U. Mass., 2001

Media Server Modeling

- Given: Server Configuration and Client Characteristics
- Objective: predict service time of the most heavily loaded disk
- Main steps:
  - Estimate number of blocks access from disk by single client
  - Estimate total number of blocks accessed from disk
  - Estimate number of blocks accessed from most heavily loaded disk
  - Compute the service time of most heavily loaded disk

Model Source: Shenoy Prashant, U. Mass, 2001

Model

- Use frame size distribution to determine the total number of blocks accessed (C_i) from the array by a client (let $P(C_i = k) = b_i^k$)
- Number of blocks accessed from disk j by client i (X_i^j)
  - Property: A request is equally likely to start on any of the D disks

$$\Rightarrow P(X_i^j = 1) = \frac{b_i^1}{D} + \frac{2 \cdot b_i^2}{D} + \cdots + \frac{D \cdot b_i^D}{D}$$

$$P(X_i^j = k) = \sum_{m=1}^{D} b_i^{(k-1)D+m} \cdot \frac{m}{D} \quad (k = 1, 2, 3...)$$

- Total number of blocks accessed from disk j by n clients

$$Y^j = \sum_{i=1}^{n} X_i^j$$

Model

- Number of blocks accessed from the most heavily loaded disk

$$Y^{\max} = \max(Y^1, Y^2, ..., Y^n)$$

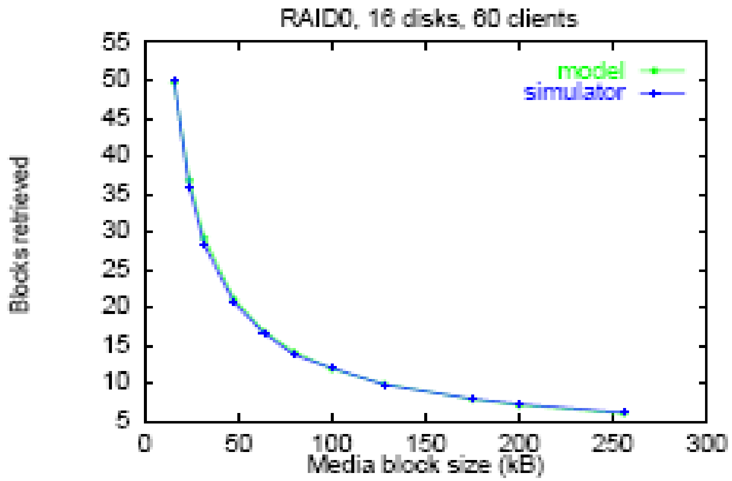
- Service time of the most heavily loaded disk

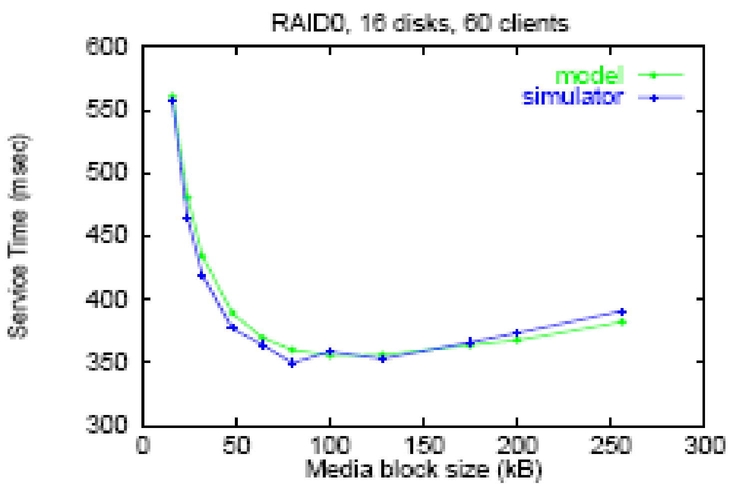
$$\tau = E(Y^{\max}) * (t_s + t_r + t_x)$$

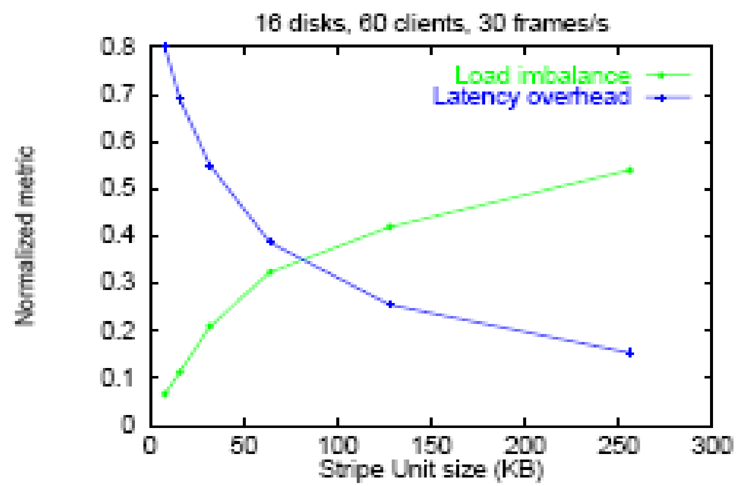
where

t_s	:	seek time to access a block
t_r	:	rotational latency to access a block
t_x	:	transfer time of a block

Validation of Model



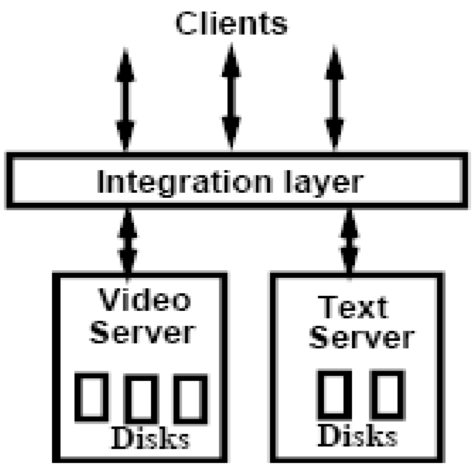


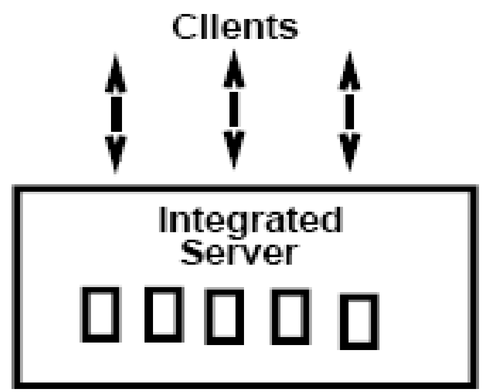


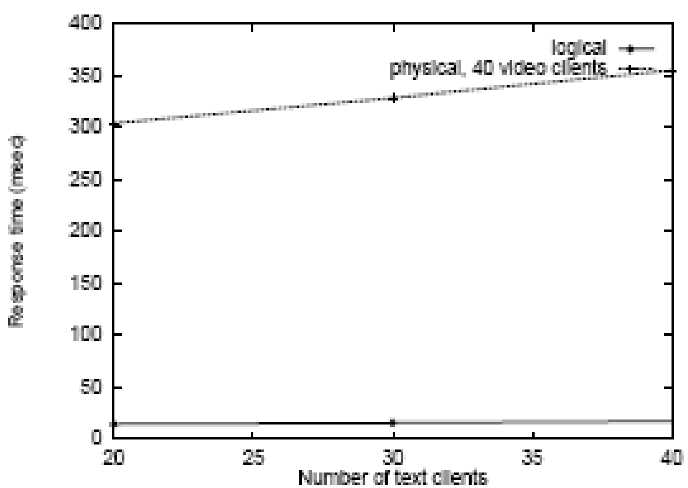
Example Multimedia File System (**Symphony**)

- *Source: P. Shenoy et al, "Symphony: An Integrated Multimedia File System", SPIE/ACM MMCN 1998*
- System out of UT Austin
- **Symphony's Goals:**
  - Support real-time and non-real time request
  - Support multiple block sizes and control over their placement
  - Support variety of fault-tolerance techniques
  - Provide two level metadata structure that all type-specific information can be supported

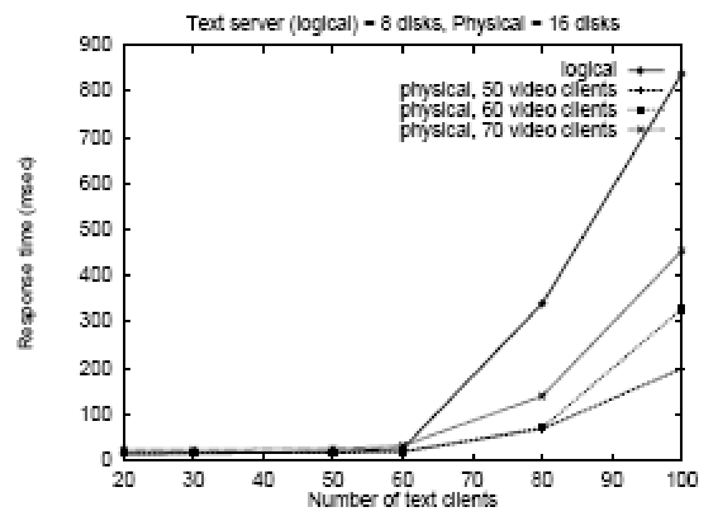
Design Decisions





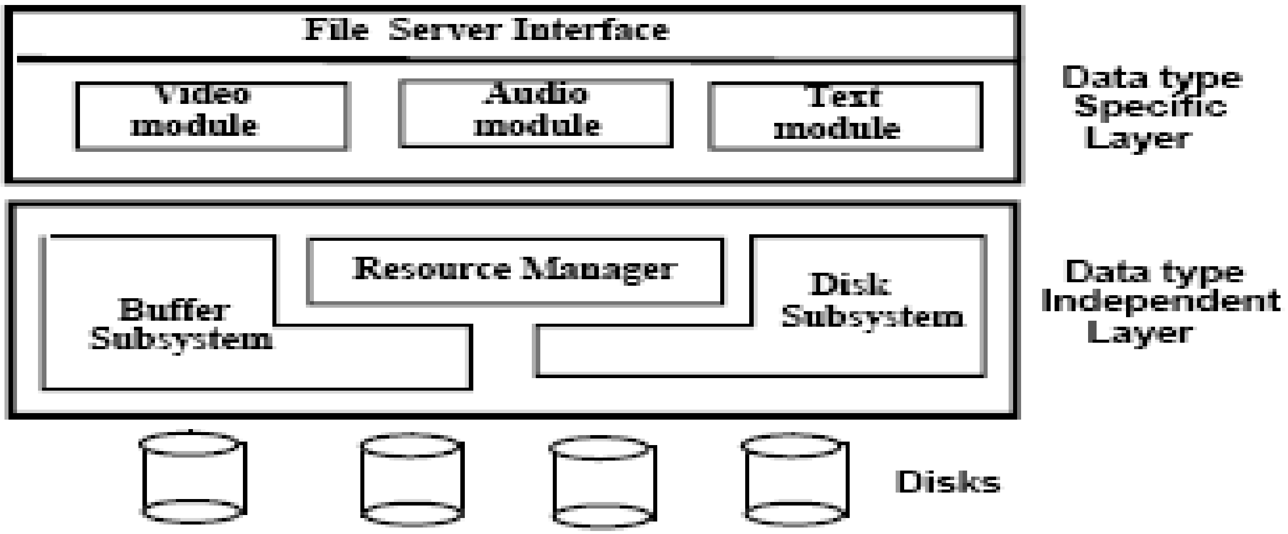


(a) SCAN



(b) Symphony disk scheduling algorithm

Two Level Symphony Architecture

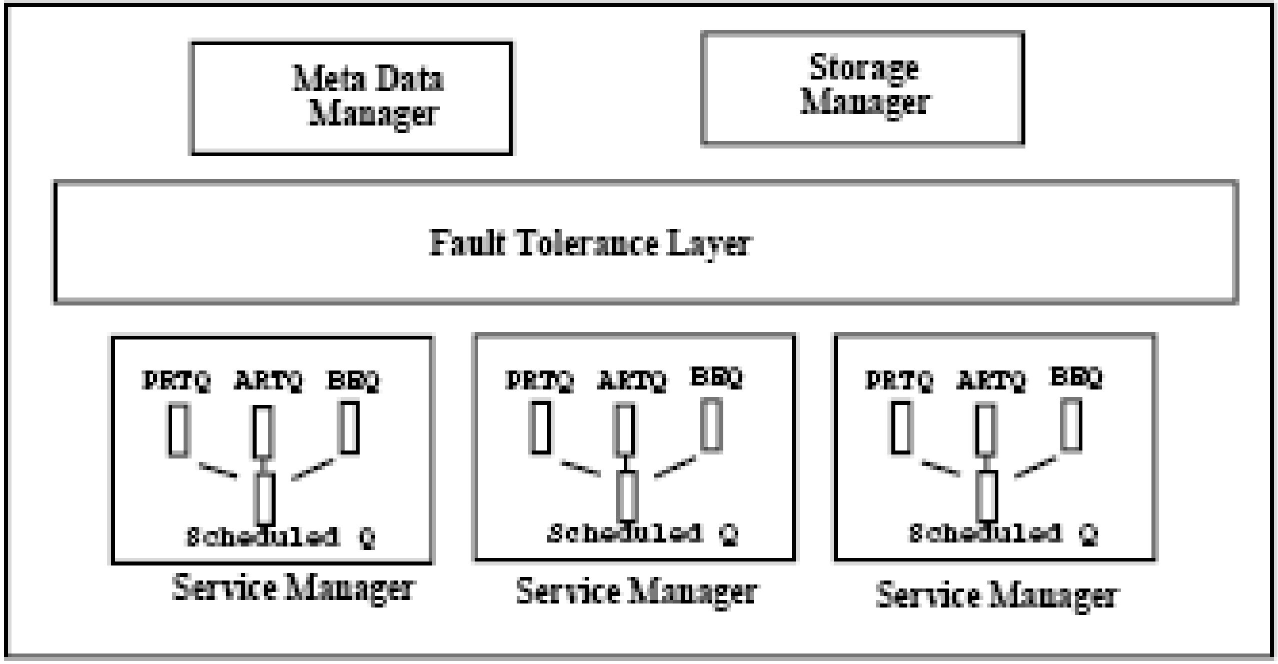


Resource Manager:

• Disk Schedule System (called Cello) that uses modified **SCAN-EDF** for RT Requests and **C-SCAN** for non-RT requests as long as deadlines are not violated

• Admission Control and Resource Reservation for scheduling

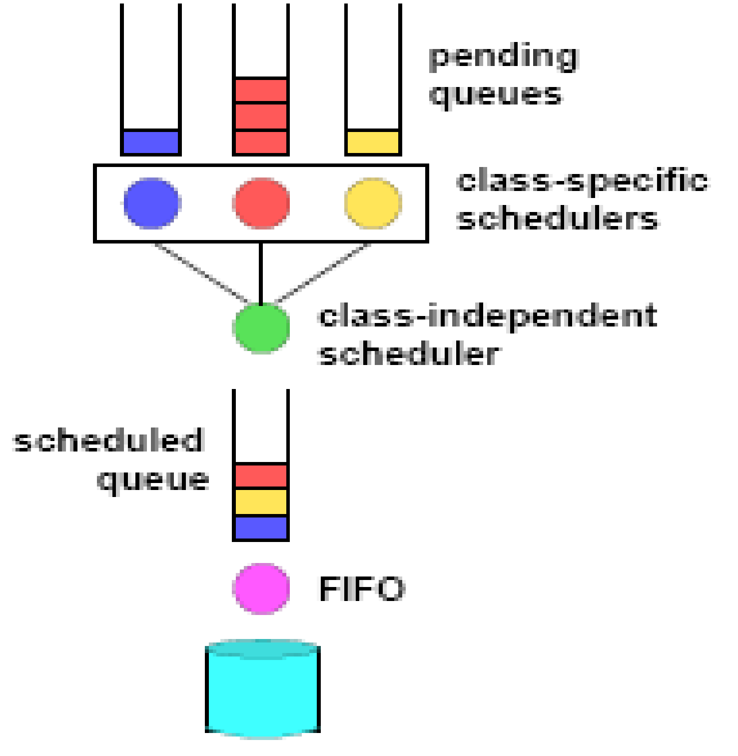
Disk Subsystem Architecture



Service Manager : supports mechanisms for efficient scheduling of best-effort, aperiodic real-time and periodic real-time requests
Storage Manager: supports mechanisms for allocation and de-allocation of blocks Of different sizes and controlling data placement on the disk
Fault Tolerance layer: enables multiple data type specific failure recovery techniques
Metadata Manager: enables data types specific structure to be assigned to files

Cello Disk Scheduling Framework

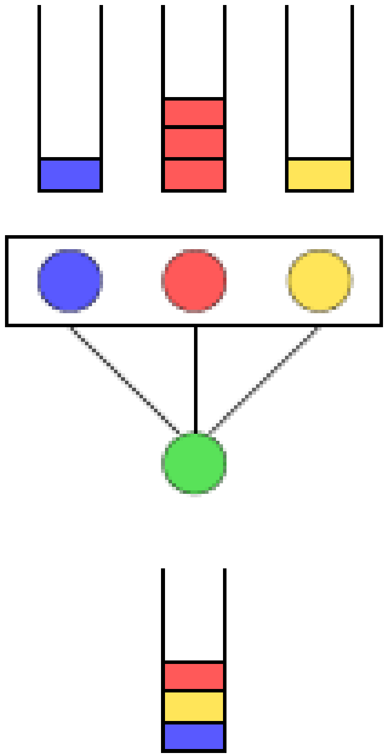
- Class-independent scheduler
  - Coarse grain bandwidth allocation to classes
  - Determines *when* and *how many* requests to insert
- Class-specific schedulers
  - Fine grain interleaving of requests
  - Create a schedule that meets request needs
  - Determine *where* to insert requests



Source: Prashant Shenoy, 2001

Class-Independent Scheduler

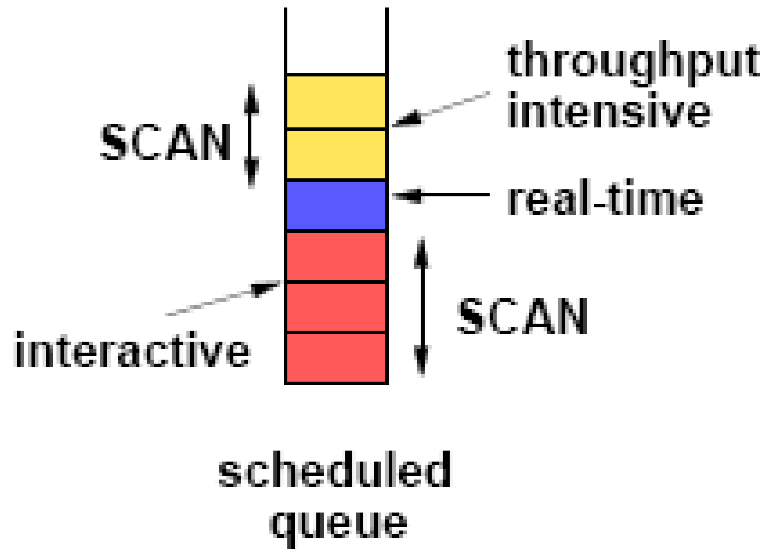
- Assign weights to each class
  - Allocate bandwidth in proportion to its weight
  - Time allocation versus byte allocation
- Algorithm:
  - Invoke a class specific scheduler for a request
  - Insert request at specified position if
    - class has sufficient unused allocation
    - total used allocation $\leq$ interval length
  - Update used allocation
  - Reallocate unused allocation to classes with pending requests



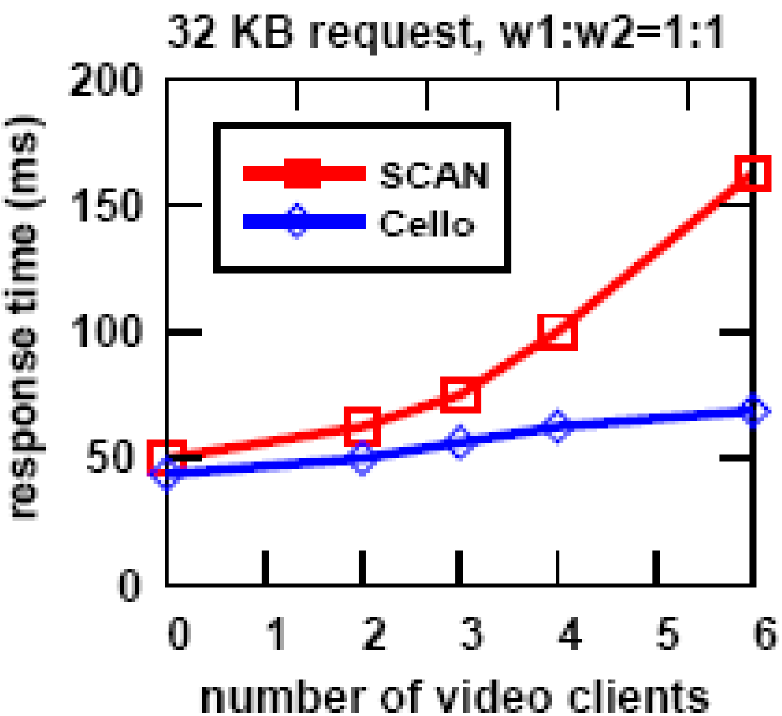
Source: Prashant Shenoy, 2001

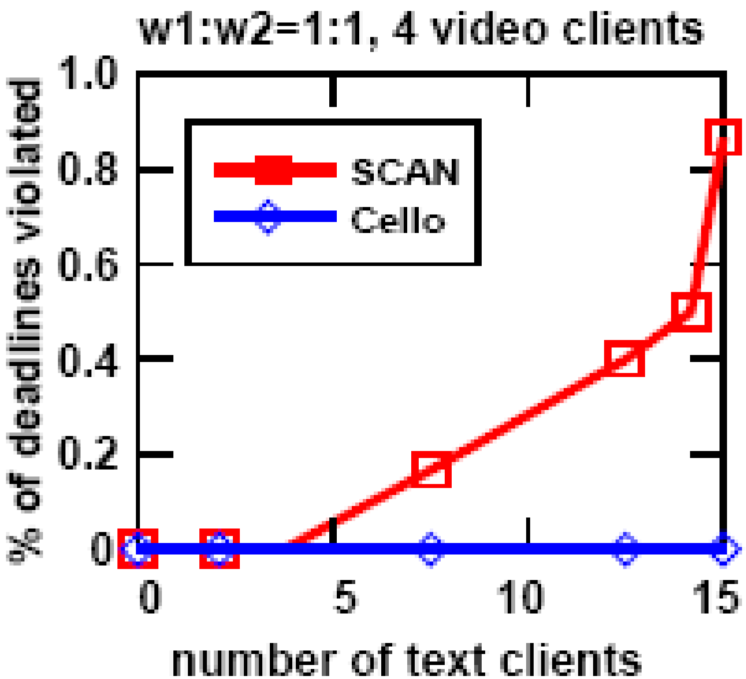
Class-Specific Schedulers

- Determine insert position based on
  - Requirements of requests (e.g., deadlines)
  - State of the scheduled queue
- Interactive best-effort
  - Insert using *slack-stealing* [Lehoczky92]
- Throughput-intensive best-effort
  - Insert at tail in SCAN order
- Real-time
  - Insert in SCAN-EDF order



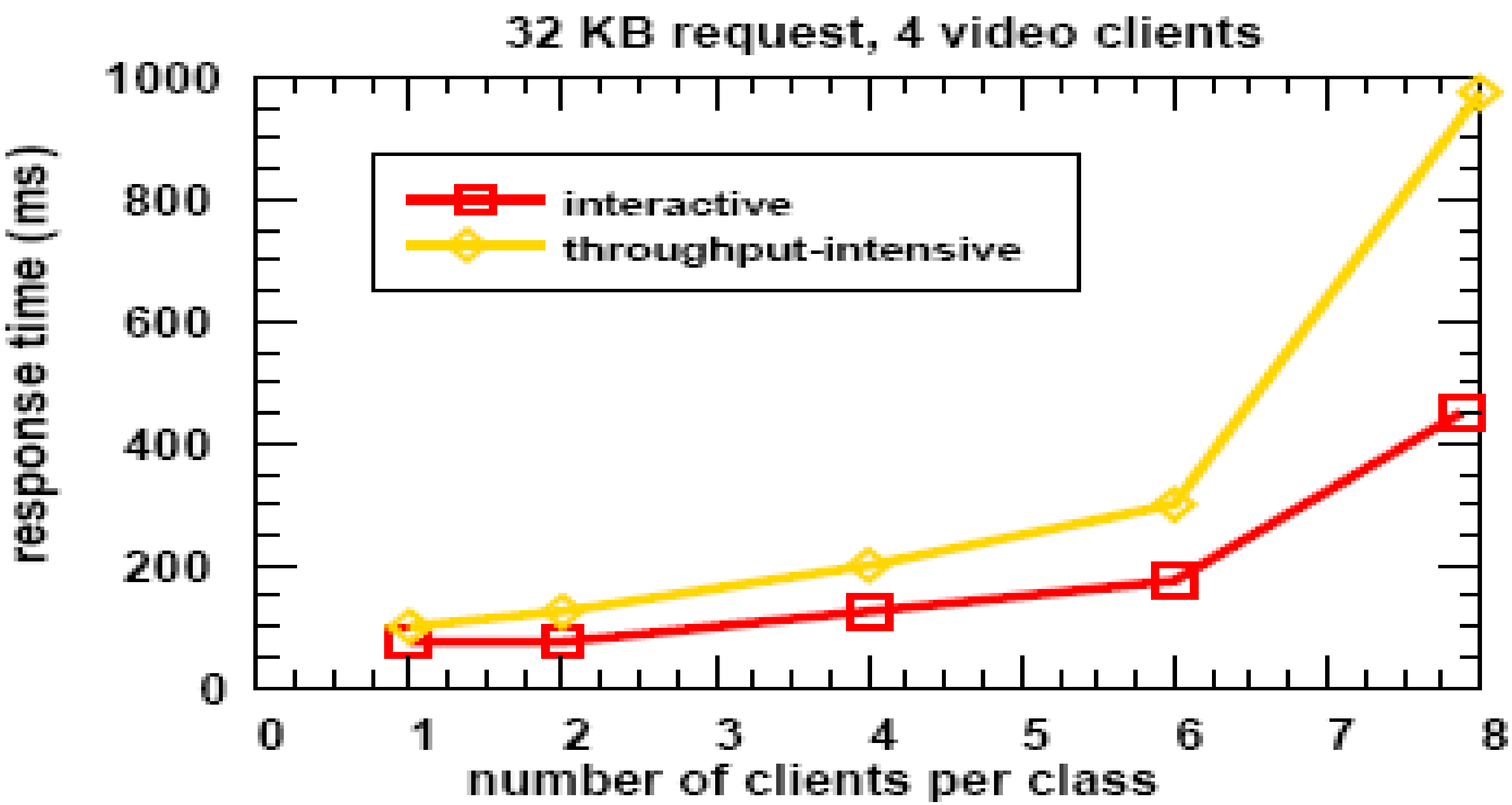
Validation: Symphony's scheduling system (Cello)





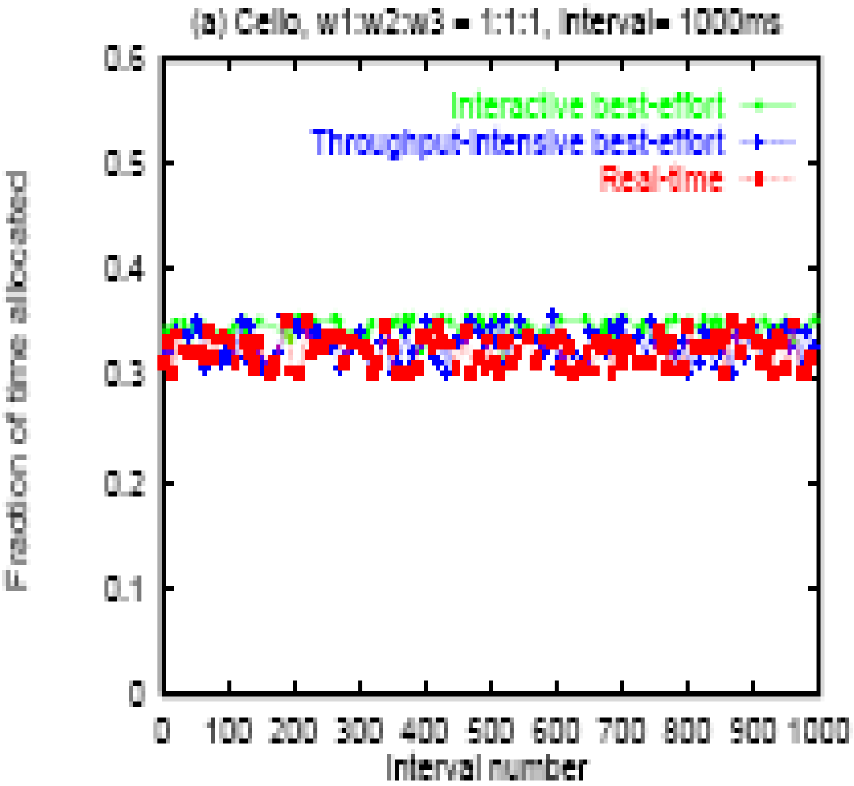
Source: Shenoy Prashant, 2001

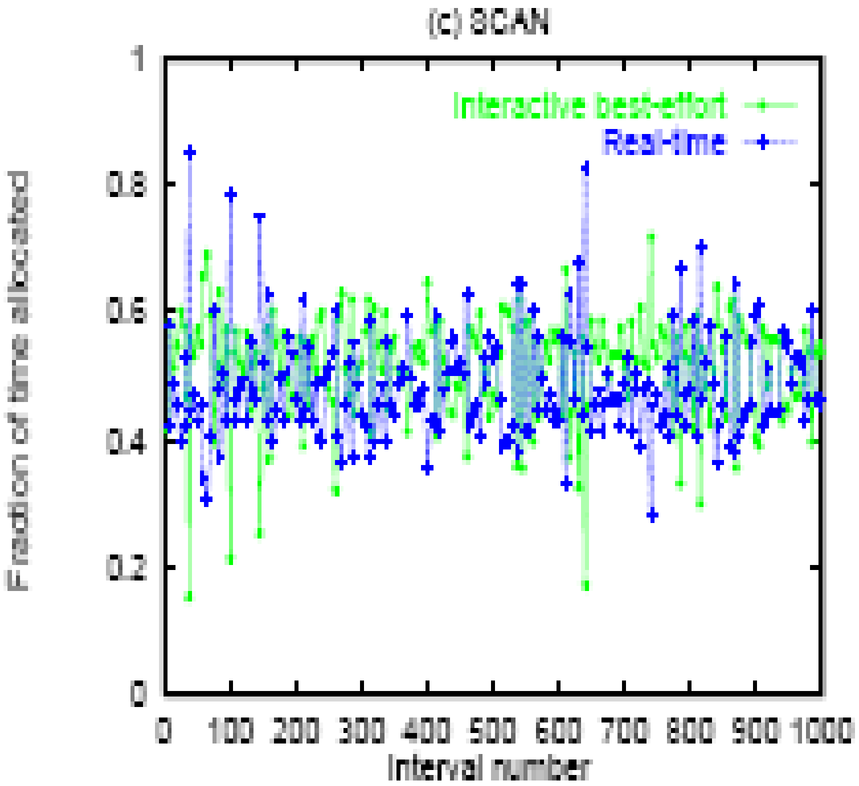
Cello provides better response time to interactive requests



Source: Shenoy Prashant, 2001

Cello achieves predictable disk BW allocation





Source: Shenoy Prashant, 2001

Buffer Subsystem

- Enable multiple data types specific caching policies to coexist
- Partition cache among various data types and allow each caching policy to independently manage its partition
- Maintain two buffer pools:
  - a pool of de-allocated buffers
  - pool of cached buffers.
    - Cache pool is further partitioned among various caching policies
    - Buffer that is least likely to be accessed is stored at the head of the list.
    - Examples of caching policies for each cache buffer: LRU, MRU

Buffer Subsystem (Protocol)

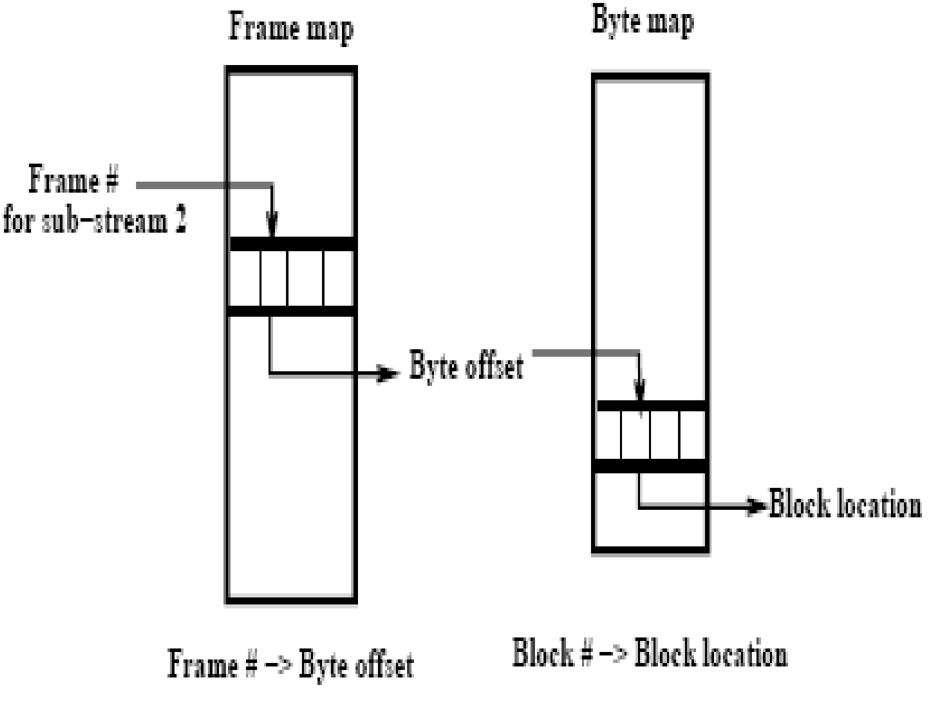
- Receive buffer allocation request
- Check if the requested block is cached.
  - If yes, it return the requested block
  - If cache miss, allocate buffer from the pool of de-allocated buffers and insert this buffer into the appropriate cache partition
- Determine (Caching policy that manages individual cache) position in the buffer cache
  - If pool of de-allocated buffers falls below low watermark, buffers are evicted from cache and returned to de-allocated pool
  - Use TTR (Time To Reaccess) values to determine victims
    - TTR – estimate of next time at which the buffer is likely to be accessed

Video Module

- Implements policies for placement, retrieval, metadata management and caching of video data
- Placement of video files on disk arrays is governed by two parameters: block size and striping policy.
  - supports both fixed size blocks (fixed number of bytes) and variable size blocks (fixed number of frames)
  - uses location hints so as to minimize seek and rotational latency overheads
- Retrieval Policy:
  - supports periodic RT requests (server push mode) and aperiodic RT requests (client pull mode)

Video Module (Metadata Management)

- To allow efficient random access at byte level and frame level, video module maintains two-level index structure
  - First level of index , referred to as frame map, maps frame offset to byte offset
  - Second level, referred to as byte map, maps byte offset to disk block locations



Caching Policy

- Interval-based caching for video module
- LRU caching for text module

Conclusion

- The data placement, scheduling, block size decisions, caching are very important for any media server design and implementation.