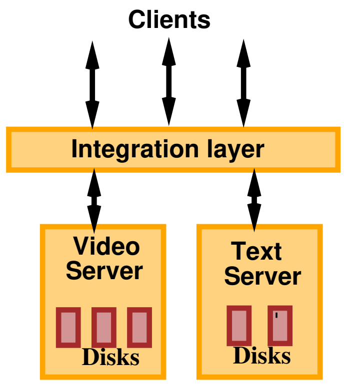
Designing Integrated Multimedia File Systems

Need for Integrated File Systems

- Different data types have significantly different characteristics
  - Textual data: aperiodic, small reads and writes, non real-time
  - Continuous media: periodic, sequential, large data rates, real-time requirements
- Most existing file systems are optimized for a single data type
  - Conventional (e.g., UNIX) file systems: optimized for text
  - Continuous media servers: optimized for audio and video
- **Conclusion:** Develop an integrated file system that efficiently supports multiple data types

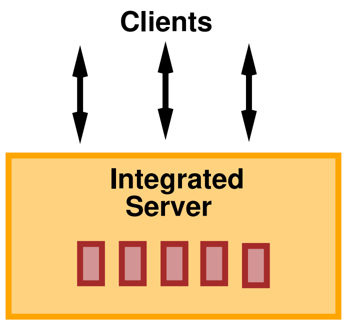
Design Philosophy: Logical Integration

- Use a separate file system for each data type, and design an integration later for providing unified access
- **Advantages:**
  - Does not require any modifications to the individual file servers ⇒simple to implement
  - Static partitioning of resources ⇒no interference
- **Disadvantages:** Under-utilization of resources due to the
  - Difficulty in altering the resource partitioning dynamically
  - Mismatch in the bandwidth and space requirements of objects



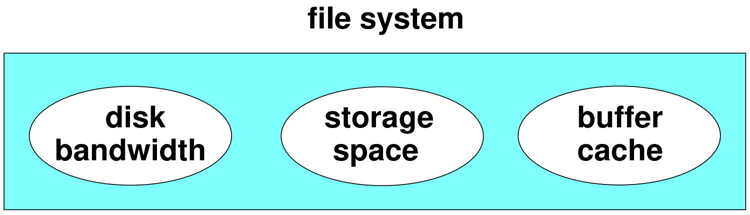
Design Philosophy: Physical Integration

- Share storage resources among all the data types
- **Advantages:**
  - Co-locating objects with different bandwidth-space requirements ⇒higher utilization
  - On-demand resource allocation ⇒easily accommodate dynamic changes in access patterns
- **Disadvantages:**
  - Interference among service classes optimizes for different data types



Integrated File System Design: Fundamentals

- File system manages three resources



- Resource management techniques
  - Disk bandwidth: disk scheduling algorithms
  - Storage space: placement and failure recovery
  - Buffer cache: caching techniques

Managing Disk Bandwidth

- Applications have different service requirements
  - Interactive applications: low average response times
  - Throughput-intensive applications: high average throughput
  - Real-time applications: performance guarantees
- Existing disk scheduling algorithms are optimized for single class of service
  - Best-effort: SCAN, SATF, ...
  - Real-time: SSEDO, SCAN-EDF, FD-SCAN, ..
- Requirements:
  - Export multiple classes of service (real-time, interactive, ...)
  - Disk schedulers that align the service provided with application needs

Managing Storage Space

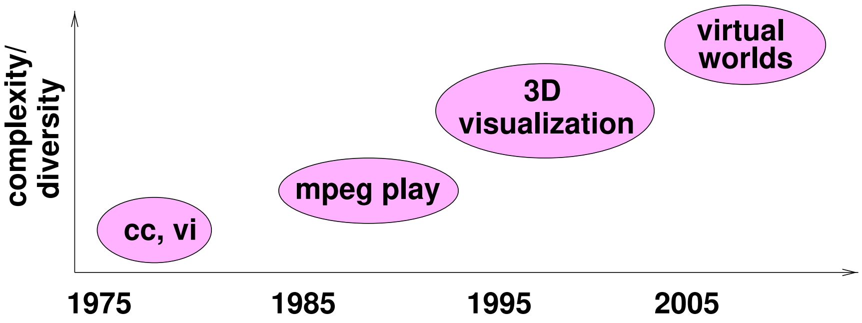
- Placement policies governed by:
  - Data characteristics
  - Application requirements
- Examples:
  - Continuous media applications: large data rates
    - large block size (64–128KB) desirable
  - Conventional applications: small object sizes
    - small block size (4–8KB) desirable
- Single placement policy unsuitable for all data types
- Requirements:
  - Support multiple data type specific placement policies
  - Employ mechanisms to enable their coexistence

Managing Buffers

- Caching policy governed by access characteristics
- Examples:
  - Continuous media applications: sequential access
    - LRU ineffective, Interval Caching suitable
  - Conventional applications: locality of reference
    - LRU caching policy suitable
- No single policy suitable for all classes
- Requirements:
  - Support multiple data type specific caching policies
  - Employ mechanisms that enable them to share the buffer cache

Architectural Requirements

- Increasing complexity and diversity of applications

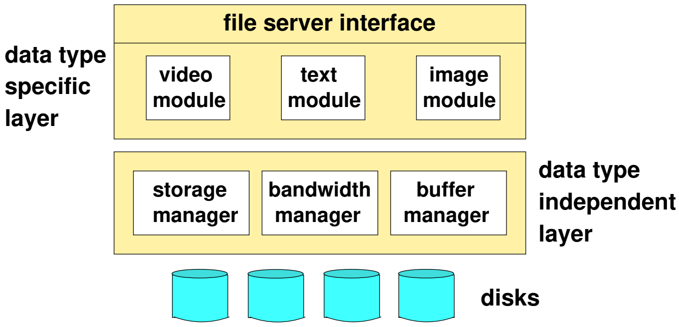


- Efficiently support present and future applications
- Requirement:
  - File system must be extensible and customizable

Requirements Summary

- **Key principle:** "One size does not fit all"
- Support multiple data type specific policies
  - Support multiple classes of service
  - Align the service provided with application needs
  - Support multiple policies for placement and caching
  - Employ mechanisms to enable their coexistence
- File system must be extensible and customizable

Symphony Architecture



- **Data type independent layer**
  - Consists of mechanisms that implement core file system functionality such as disk scheduling, storage management, buffer management, etc.
- **Data type specific layer**
  - Consists of modules that implement data type specific placement policies, retrieval policies, caching policies, etc.

Data Type Independent Layer

- Storage manager
  - Placement of data and meta-data blocks on disks
  - Fault tolerance
- Bandwidth manager
  - Retrieval architectures
  - Disk bandwidth allocation
  - Resource reservation
- Buffer manager
  - Buffering and caching

Disk Scheduling: Introduction

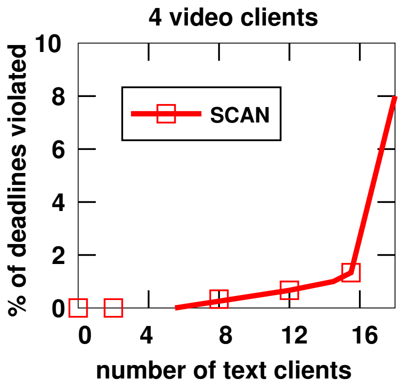
- Requirement: Integrated file systems must support multiple application classes
- Examples:
  - Interactive best effort applications
    * editors, compilers
  - Throughput-intensive best-effort applications
    * http and ftp servers
  - Soft real-time applications
    * virtual reality, video players
  - Hard real-time applications
    * control systems

Existing Disk Scheduling Algorithms

- Optimized for a single class of applications
- Best-effort disk scheduling algorithms
  - SCAN, SATF [Jacobson91, Seltzer90, Teorey72, Worthington94]
    * schedule requests based on relative positions on disk
    * minimize seek time and relational latency overheads
- Real-time disk scheduling algorithms
  - SSEDO, FD-SCAN, SCAN-EDF [Abbott90, Chen91, Reddy93]
    * start from an EDF schedule
    * reorder requests to reduce seek time and relational latency
- Are existing disk scheduling algorithms adequate?

Existing Disk Scheduling Algorithms: Limitations

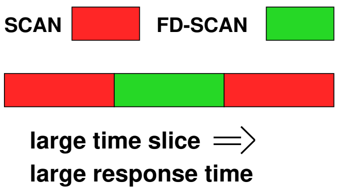
- Poor performance for mixed workloads
  - Example: SCAN
    - Does not take requirements of requests into account (e.g., deadlines)
    - Unsuitable for real-time applications

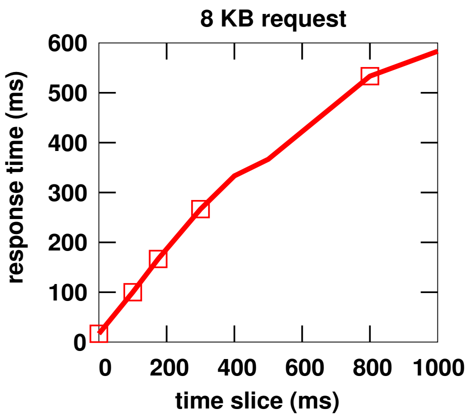


- Priority-based scheduling can lead to starvation

Supporting Multiple Application Classes

- Coarse-grain time slicing

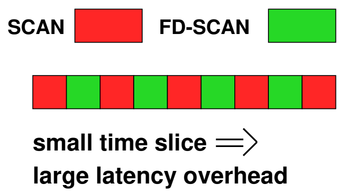


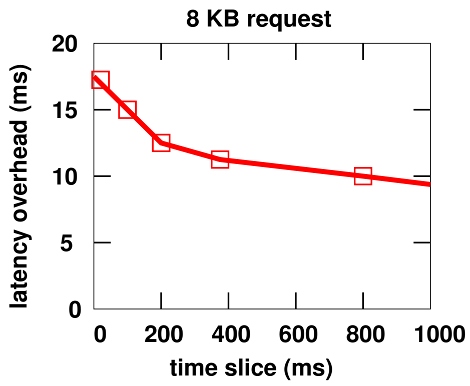


- Fine-grain time slicing: hierarchical schedulers
  - CPU lottery scheduling [Waldspurger94]
  - Network: SFQ [Goyal96], H-WF^{2}Q+ [Bennett96]

Why Is Servicing Multiple Classes Hard?

- Hierarchical schedulers assume a fixed context switch overhead
  - Disk is fundamentally different from other resources
    - variable context switch time (seek and rotational latency)
    - reduce latency overhead to maximize throughput
  - Fine-grain allocation $\Rightarrow$ large latency overhead



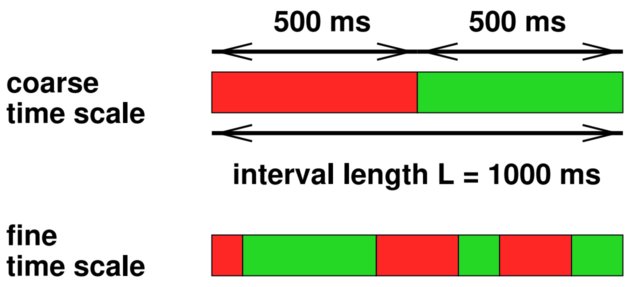


Disk Scheduling Requirements

- Export multiple classes of service (interactive, real-time, ...)
- Align the service provided with application needs
- Reduce seek time and rotational latency overheads
- Protect application classes from each other
- Computationally efficient

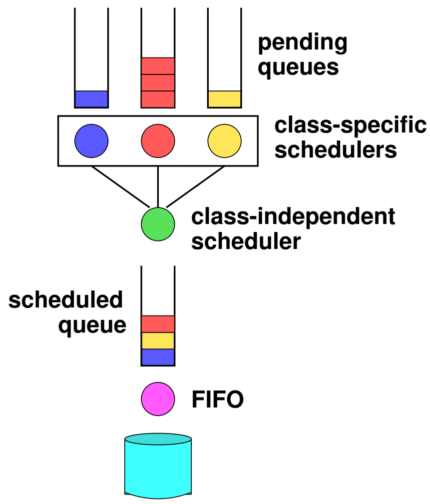
Cello: Key Principles

- Unique feature: Disk bandwidth allocation at two time-scales
  - Coarse time-scale bandwidth allocation: Allocate a certain fraction of disk bandwidth to each class
  - Fine time-scale bandwidth allocation: Determine a schedule that
    - Reduces seek time and rotational latency
    - Aligns the service with application needs



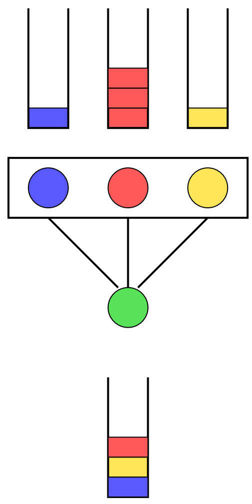
Cello Disk Scheduling Framework

- 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



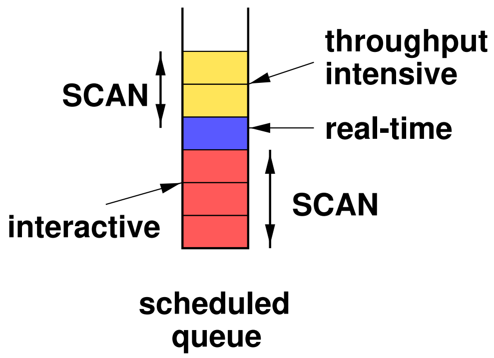
Class-Independent Scheduler

- 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

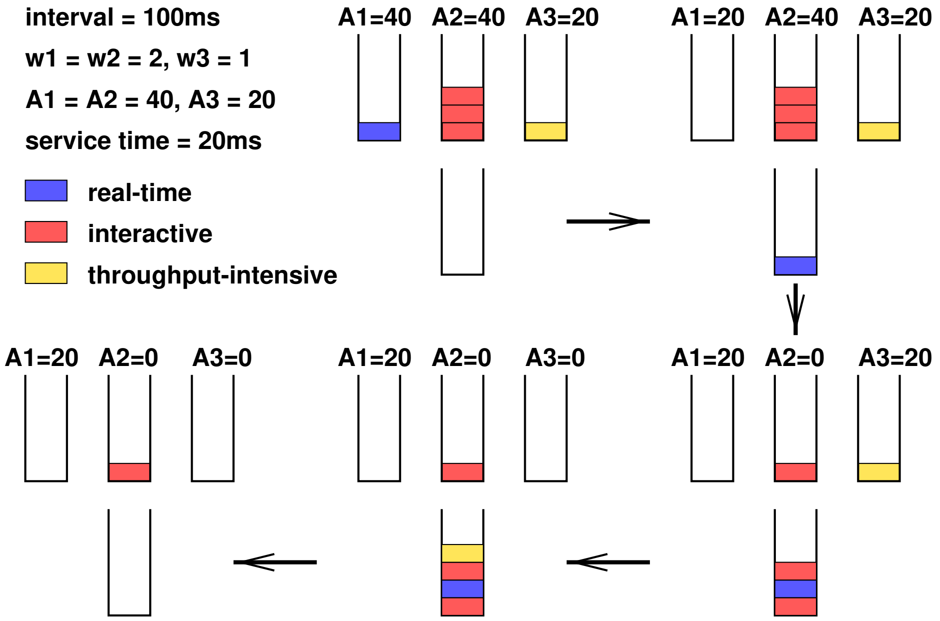


Class-Specific Schedulers

- 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

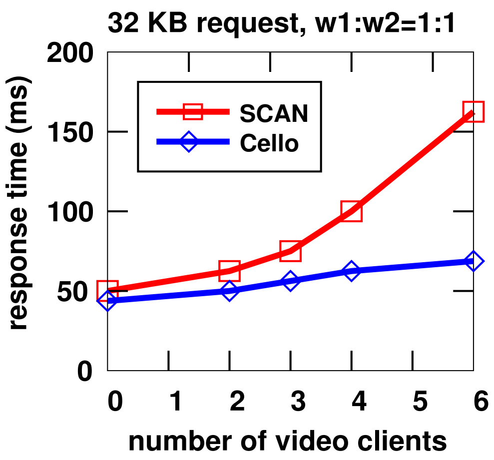


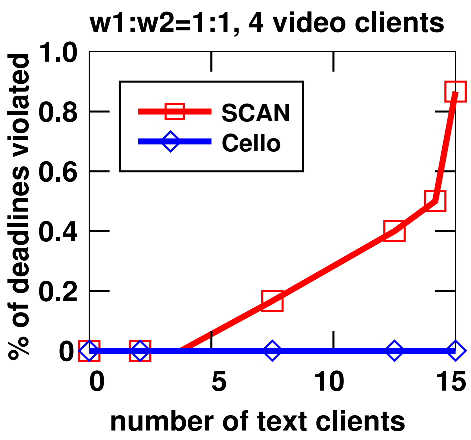
Cello: Example



Performance Evaluation (1)

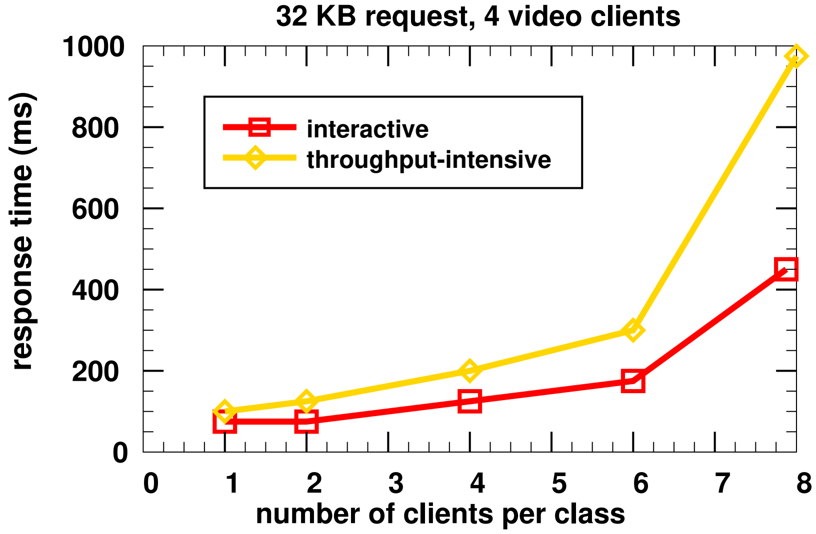
- Cello protects application classes from each other and aligns the service to application needs





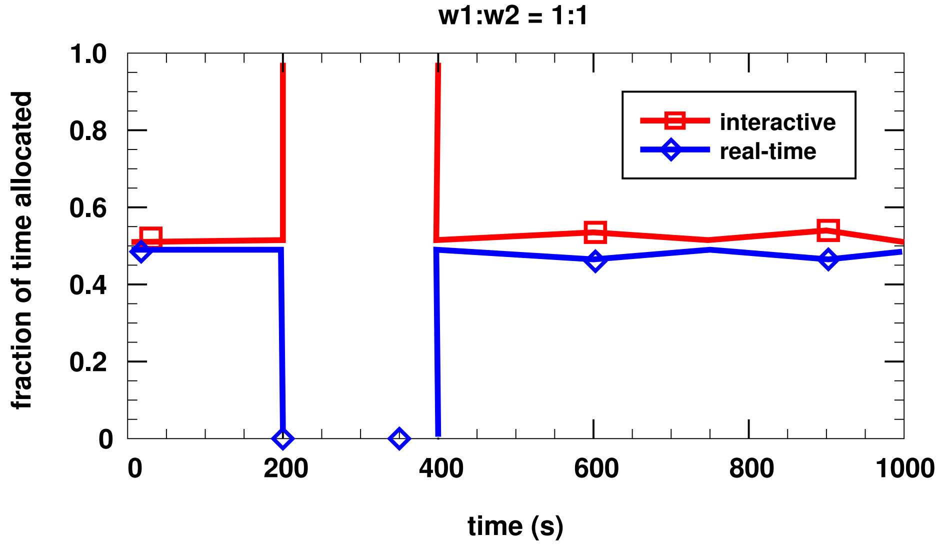
Performance Evaluation (2)

- Cello provides better response times to interactive requests



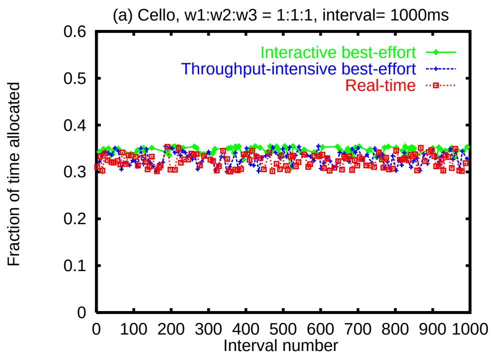
Performance Evaluation (3)

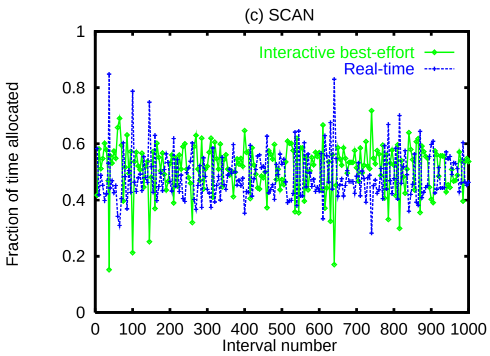
- Cello can dynamically reassign unused bandwidth



Performance Evaluation (4)

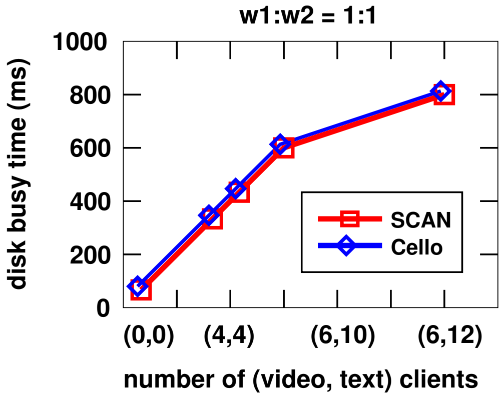
- Cello achieves predictable disk bandwidth allocation





Performance Evaluation (5)

- Cello incurs comparable seek and rotational latency overhead



- Cello is computationally efficient

	SCAN		Cello	
	average case	worst case	average case	worst case
total insert time	129 μs	7.4 ms	180 μs	12 ms

Data Type Independent Layer

- Storage manager
  - Placement of data and meta-data blocks on disks
  - Fault tolerance
- Bandwidth manager
  - Retrieval architectures
  - Disk bandwidth allocation
  - Resource reservation
- Buffer manager
  - Buffering and caching

Storage Manager

- **Objective**: Enable the coexistence of multiple stripe unit sizes, degrees of striping, and striping policies
- Multiple stripe unit sizes
  - Define a base block size at file system creation time
  - Construct a stripe unit using a sequence of continuous base blocks
- Multiple degrees of striping and striping policies
  - Provide control over placement of blocks using location hints
  - Export the presence of multiple disks

Fault Tolerance Layer

- **Objective**: Mask disk failures and enable multiple failure recovery policies to coexist
- Approach: Separate off-line rebuild from on-line reconstruction
- Off-line rebuild using parity information
- Multiple on-line reconstruction techniques:
  - Perfect reconstruction: use parity-based reconstruction
  - Approximate reconstruction at clients: disable parity-based reconstruction

Disk Bandwidth Manager

- **Objective**: Schedule requests with different QoS requirements
- Supports three service classes
  - Best-effort, periodic real-time, and aperiodic real-time
- Supports two service architectures
  - Server-push and client-pull
- Uses a QoS-differentiating disk scheduling algorithm
  - Delay real-time requests until their deadlines
  - Use available slack to service best-effort requests
  - Prevent starvation by guaranteeing a minimum amount of bandwidth to each service class

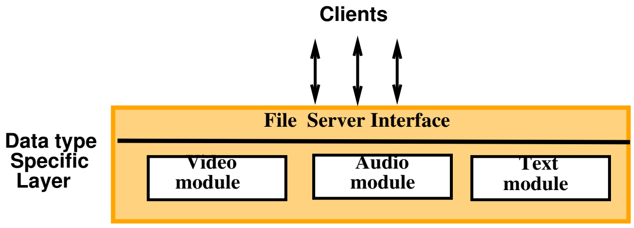
Resource Reservation

- **Objective**:
  - Partition resources among various service classes
  - Use admission control, if necessary, to meet QoS requirements of each service class
- Admission control algorithms
  - Deterministic
    - Uses worst-case assumptions to estimate resource availability
    - Provides hard guarantees
  - Statistical
    - Uses probability distributions to estimate resource availability
    - Provides probabilistic guarantees

Buffer Manager

- **Objective**: Support multiple data type specific caching policies
- Partition cache among data types
  - Partitioning can be static or dynamic
- Allow each caching policy to independently manage its cache partition
  - Caching policies are implemented by the data type specific layer
  - Buffer deallocation $\Rightarrow$ invoke appropriate caching policy

Data Type Specific Layer

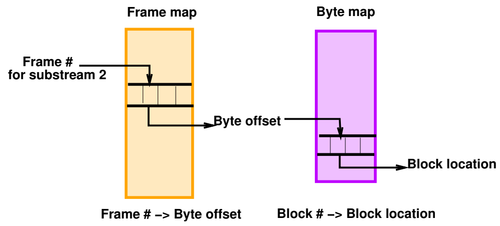


- Consists of a set of modules that implement data type specific policies
- Exports a file server interface
  - Methods for creating, deleting, reading, and writing files

Video Module

- Retrieval Policy
  - Supports both server-push and client-pull
  - Server-push requests are serviced as periodic real-time requests
  - Client-pull requests are serviced as aperiodic real-time requests
- Placement Policy
  - Supports fixed-size and variable-size blocks
  - Optimizes placement of multi-resolution video streams
    * Facilitates retrieval of streams at the desired level of resolution
    * Uses location hints so as to minimize seek and rotational latency overheads

Video Module (Cont'd.)



- Meta Data Management
  - Allows byte-level indexing as well as frame-level indexing
  - Uses a two level index
    - Level 1 (frame map) : Maps frame offsets to byte offsets
    - Level 2 (byte map): Maps byte offsets to disk block locations
- Caching Policy: Uses Interval Caching to cache blocks

Text Module

- Similar to conventional UNIX file systems
- Retrieval policy: supports client-pull, best-effort requests
- Placement policy: uses round-robin striping of fixed-size blocks
- Meta Data Management: uses a byte map (similar to UNIX inode)
- Caching policy: uses LRU to cache blocks

Related Work

- Text file systems [Bach86, McKusick84, Leffler89,...]
- Video-on-demand servers [Tobagi93, Vernick96,...]
- Integrated file systems
  - CMFS [Anderson92], Fellini [Ozden97]
  - XFS [Holten95], IBM Tiger Shark [Haskin96]
  - MARS [Buddhikot98], MMFS [Niranjan97]
  - RT-FS [Rajkumar98]
- Extensible operating systems
  - Exokernel [Engler95]
  - SPIN [Bershad95]