A File System for Continuous Media

DAVID P. ANDERSON
Sonic Solutions

YOSHITOMO OSAWA
Sony Corporation
and
RAMESH GOVINDAN
University of California at Berkeley

The Continuous Media File System, CMFS, supports real-time storage and retrieval of continuous media data (digital audio and video) on disk. CMFS clients read or write files in ‘‘sessions’’, each with a guaranteed minimum data rate. Multiple sessions, perhaps with different rates, and non-real-time access can proceed concurrently. CMFS addresses several interrelated design issues: real-time semantics of sessions, disk layout, acceptance test for new sessions, and disk scheduling policy. We use simulation to compare different design choices.

Categories and Subject Descriptors: D.4.3 [Operating Systems]: File System Management – *access methods, file organization*; D.4.8 [Operating Systems]: Organization and Design – *real-time systems*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Multimedia, disk scheduling

1. INTRODUCTION

Current disk drives have raw data rates of 5 to 10 million bits per second (Mbps) or more. Such rates suffice for many forms of digital audio and motion video (*continuous media*, or CM) data: audio data rates are from 8 Kbps to 1.4 Mbps, while compressed video ranges from one to several Mbps. However, when a disk is accessed via a general-purpose file system, the data rates seen by clients are generally lower and may vary unpredictably.

We have developed a *Continuous Media File System* (CMFS) whose clients read and write files in ‘‘sessions’’, each with a guaranteed minimum data rate. Multiple sessions, perhaps with different data rates, can coexist. CMFS can handle non-real-time traffic concurrently with these real-time sessions.

Authors’ addresses: D.P. Anderson, 1891 East Francisco Blvd. San Rafael, CA 94901. Y. Osawa, MO Business Development Division, Storage Systems Group, Sony Corporation, 2255 Okata, Atsugi, Kanagawa 243 Japan. R. Govindan, CS Division, EECS Department, UC Berkeley, Berkeley, CA 94720.

This work was performed at the University of California at Berkeley, and was supported by NSF Infrastructure Grant CDA-8722788, NSF PYI grant CCR-86-57529, the California MICRO program, and Sun Microsystems.

To provide data rate guarantees, CMFS addresses the following interrelated issues:

- **Real-time semantics:** The CMFS client interface, described in Section 2, has flexible but well-defined real-time semantics.
- **Disk layout:** Section 4 gives the CMFS assumptions about disk layout.
- **Acceptance test:** Section 5 describes how CMFS determines if a new session can be accommodated.
- **Disk head scheduling:** Several alternative policies for ordering disk read and write operations are discussed in Sections 6 and 7.

Several broad classes of CM data servers can be envisioned: workstation file systems that handle voice mail messages as well as other data; network-accessible archives of data resources (lectures, hypermedia documents, *etc.*) for research and education; and, with the advent of B-ISDN networks, commercial information services offering movies, news, and music to hundreds or thousands of concurrent clients. High-level issues such as security, naming and indexing, and file structuring differ among these classes; we do not deal with these issues here. However, there is a common need to store and retrieve data streams with predictable real-time performance; thus the concepts and techniques of CMFS apply to each class.

CMFS is meant to serve as part of a distributed system that handles integrated audio and video. End-to-end performance guarantees cannot, of course, be achieved by disk scheduling alone. CMFS conforms to the ‘‘meta-scheduling’’ model [2], which provides a mechanism for making such guarantees. The role of CMFS in this larger context is discussed in Section 3.

2. CLIENT INTERFACE

CMFS clients access real-time files in *sessions*. Each session has a FIFO buffer for data transfer between the client and CMFS. For concreteness, assume that CMFS is part of an OS kernel, a client is a kernel process, and FIFOs are circular buffers in physical memory. Alternatives will be discussed in Section 3.1.

2.1. The Semantics of Sessions

The flow of data in a session is not necessarily smooth or periodic. Instead, session semantics are defined in terms of a ‘‘logical clock’’ that runs at a fixed rate through the FIFO, stopping if it catches up to the client’s position. CMFS promises to stay ahead of the logical clock by a given positive amount (the ‘‘cushion’’). These semantics allow CMFS to handle variable-rate files and other non-uniform access in a simple way. Because CMFS is guided by client behavior, it need not know about data timestamps or file internals.

A session is created using

```
ID = request_session(
    int direction,        /* READ or WRITE */
    FILE_ID name,
    int offset,
    FIFO* buffer,
    TIME cushion,
    int rate);

start_clock(ID);
```

If `direction` is `READ`, `request_session()` requests a session in which the given file is read sequentially starting from the given offset (henceforth assumed to be zero). If the session cannot be accepted, an error code is returned. Otherwise, a session is established and its ID is returned. `Start_clock()` starts the session’s logical clock; the client can remove up to `cushion` amount of data before calling this. The appropriate value for `cushion` depends on the client's maximum delay in handling data (see Section 3.2). The client is notified (via an RPC or exception) when the end of the file has been reached. CMFS provides a `seek()` operation that flushes data currently in the FIFO and repositions the read or write point.

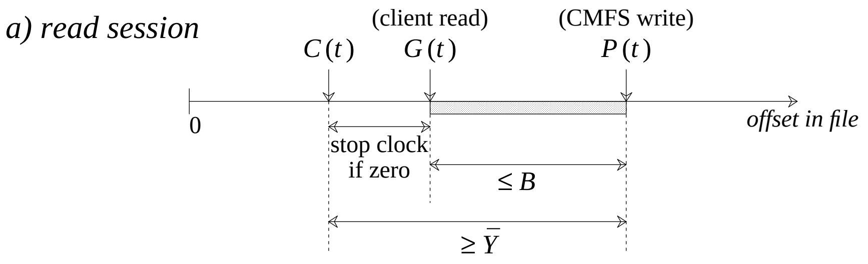
To describe the semantics of read sessions more formally, we use the following notation (see Figure 1a).

R:	the `rate` argument to `request_session()`.
$\bar{Y}$:	the `cushion` argument to `request_session()`.
t_{start}:	the time when `start_clock()` is called.
$P(t)$:	the index of the next byte to be put into the FIFO by CMFS at time t.
$G(t)$:	the index of the next byte to be removed from the FIFO by the client at time t.
$C(t)$:	the value of the logical clock at time t.
B:	the size of the FIFO buffer, in bytes.

The logical clock $C(t)$ is zero for $t = t_{start}$, and $C(t)$ increases at rate R whenever $C(t) < G(t)$. The following ''Read Session Axioms'' must hold for all $t \geq t_{start}$:

$$P(t) - G(t) \leq B \tag{1}$$

$$P(t) - C(t) \geq \bar{Y} \tag{2}$$



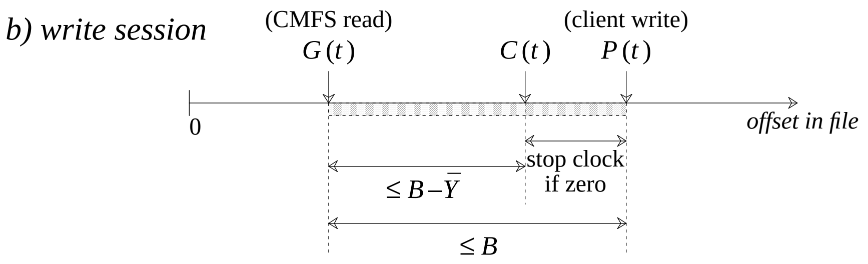


Figure 1: The semantics of read and write sessions are described in terms of a ''put pointer'' $P(t)$, a ''get pointer'' $G(t)$, and a logical clock $C(t)$. The shaded rectangles represent data in the FIFO.

$$G(t) \le P(t) \tag{3}$$

These conditions say that CMFS does not overflow the FIFO, CMFS allows the client to read ahead of the logical clock by up to $\overline{Y}$ bytes, and the client does not read beyond the write point. CMFS therefore provides a guaranteed minimum data rate, but only as long as the client keeps up with its reading. There is no upper bound on the actual data rate; the client and CMFS can in principle work arbitrarily far ahead of the logical clock.

In a write session, the client transfers data into the FIFO buffer and CMFS moves data from the buffer to disk. We describe the semantics of write sessions using the same notation as above. In this case, $P(t)$ is the index of the next byte to be inserted in the FIFO by the client, and $G(t)$ is the index of the next byte to be removed by CMFS. The logical clock $C(t)$ increases at rate R whenever $C(t) < P(t)$. The following ''Write Session Axioms'' must hold for all $t \ge t_{start}$ (see Figure 1b):

$$P(t) - G(t) \le B \tag{4}$$

$$C(t) - G(t) \le B - \overline{Y} \tag{5}$$

$$G(t) \le P(t) \tag{6}$$

These conditions say that the client does not overflow the FIFO, that CMFS removes data from the FIFO fast enough so that the client can always write ahead of the logical clock by at least $\overline{Y}$, and that CMFS does not read beyond the write point.

2.2. File Creation and Non-Real-Time Access

CMFS supports both *real-time* and *non-real-time* files. A real-time file is created using

```
create_realtime_file(
    BOOLEAN expandable,
    int size,
    int max_rate);
```

`expandable` indicates whether the file can be dynamically expanded. If not, `size` gives its (fixed) size. `max_rate` is the maximum data rate (bytes per second) at which the file is to be read or written. CMFS rejects the creation request if it lacks disk space or if `max_rate` is too high.

Non-real-time operations may be performed on either type of file. CMFS provides two non-real-time service classes: *interactive* and *background*. Interactive access is optimized for fast response, background for high throughput. There are no performance guarantees for non-real-time operations.

2.3. The Symmetry of Reading and Writing

In describing session acceptance and scheduling algorithms, the redundancy of treating read and write sessions separately can be avoided by observing the following symmetry between reading and writing. Suppose a write session S_W has fixed parameters t_{start}, B, $\overline{Y}$, and R and time-varying parameters $C_W(t)$, $P_W(t)$, $G_W(t)$. Consider an (imaginary) read session S_R having the same t_{start}, B, $\overline{Y}$ and R parameters, and for which

$$G_R(t) = P_W(t) \tag{7}$$

$$P_R(t) = G_W(t) + B \tag{8}$$

(see Figure 2). Each disk block written by CMFS in S_W advances P_W, and thus corresponds to a disk block read by CMFS in S_R.

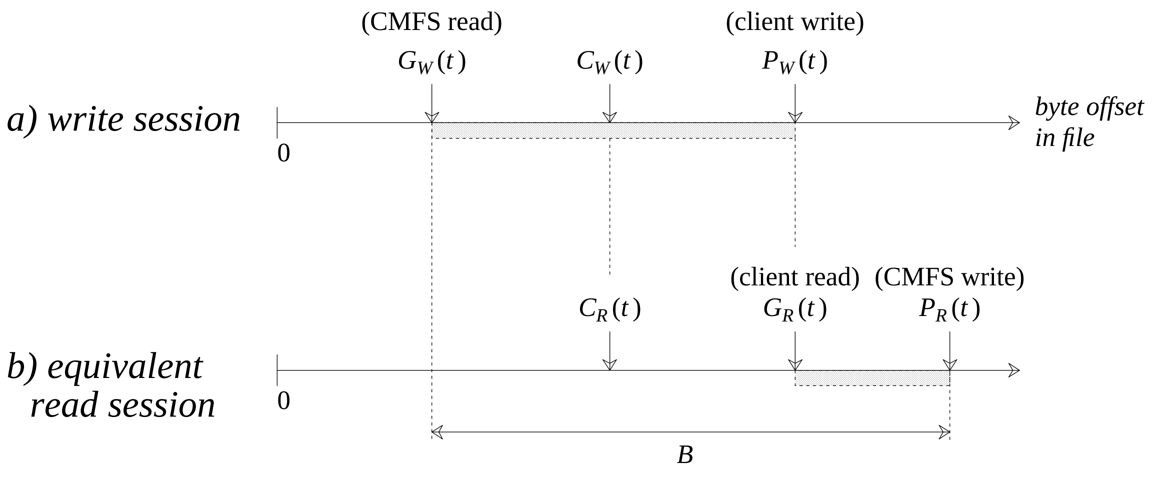


Figure 2: By interchanging empty/full and read/write, a write session (a) is transformed into a read session (b) that is equivalent with respect to scheduling.

Claim 1. *Suppose S_W is a write session, and S_R is defined as above. Then $C_R(t) = C_W(t)$ for all $t \geq t_{start}$, and S_R satisfies the Read Session Axioms (Eqs. 1, 2, and 3) if and only if S_W satisfies the Write Session Axioms (Eqs. 4, 5, and 6).*

Proof. When $G(t)$ is substituted for $P(t)$ in the definition of $C_W(t)$, the result defines $C_R(t)$. Thus the logical clock advances in S_R exactly when it advances in S_W, and at the same rate, so $C_W(t) = C_R(t)$. The equivalence of the Read and Write axioms then follows from substituting Eqs. 7 and 8 in Eqs. 1 and 2. □

So, from the point of view of scheduling in CMFS, reading and writing are essentially equivalent. The main difference is the initial condition: an empty buffer for a write session corresponds to a full buffer for a read session. In describing CMFS's algorithms for scheduling and session acceptance, we will refer only to read sessions.

2.4. Using the CMFS Interface

An idealized client might read 1 byte from the FIFO every $1/R$ seconds, beginning at the moment of session acceptance. In general, however, this uniformity is neither necessary nor desirable. For example: 1) data may be grouped into large chunks (for example, video frames) that are needed at a single moment; 2) data may have long-term rate variation (perhaps because of variable-rate compression); 3) clients may delay initial I/O to synchronize multiple sessions; 4) clients may pause and resume I/O during sessions; 5) clients may ‘‘work ahead’’, filling up intermediate buffers to improve overall performance.

The CMFS interface accommodates these requirements. To show this, we start by defining a notion of temporal data with variable but bounded rate.

Definition: A *bounded-rate file* F has parameters R and E, and the ith byte of F has a timestamp $T(i) \geq 0$ such that $T(i) \leq T(j)$ for $i < j$, and

$$i - j \leq R(T(i) - T(j)) + E \qquad (9)$$

for all $i > j$ (see Figure 3). In other words, the amount of data in a time interval of length T is at most $RT + E$. (In a digital video file, for example, R is the maximum long-term data rate, and E is the maximum number of bytes per frame.) The timestamps may be explicit (embedded in the data) or implicit.

Definition: Suppose a client reads a bounded-rate file F. We say that the file is *read in real time starting at* t_0 *with buffer size* N if 1) byte i is read (*i.e.*, removed from the FIFO) before $t_0 + T(i)$ and 2) at any time t, no more than N bytes i such that $T(i) > t - t_0$ have been read.

Intuitively, this means that the client reads the file fast enough to get the data on time, but slow enough so that only N bytes of additional buffer space are needed. The CMFS interface allows a client to read a bounded-rate file in real time using limited buffer space:

Claim 2. *Suppose a client creates a session reading a bounded-rate file* F *with parameters* R *and* E*, and the session begins at time 0. Then it is possible for the client to read* F *in real time starting at* E/R *with buffer size* E*.*

The proof is given in an Appendix.

The implicit flow control provided by the CMFS interface and can accomplish other goals as well:

- A client can pause a session by simply stopping the removal of data from the FIFO; the logical clock will stop soon thereafter. Data rate and buffer-requirement guarantees will remain valid after the client resumes reading. (The pause/resume is equivalent to shifting the timestamps of the remaining data.)

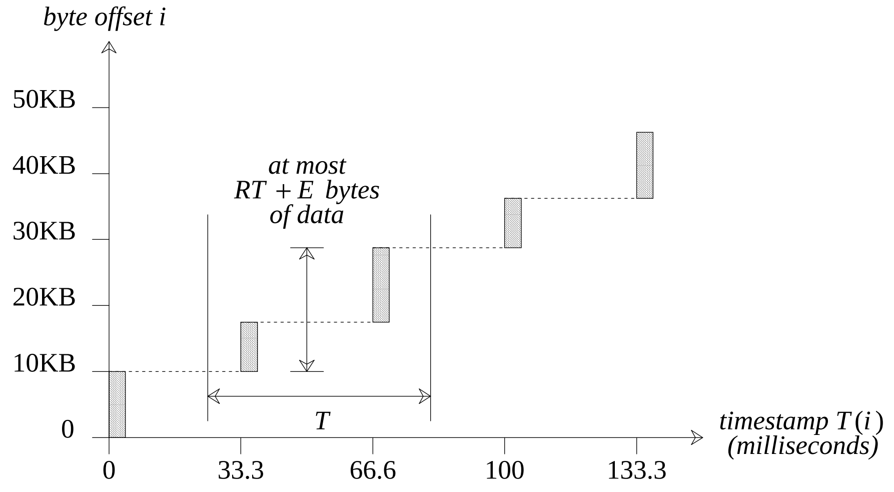


Figure 3: A *bounded-rate file* contains timestamped data. This example represents a file of 30 frames/second video data with a varying number of bytes per frame; each frame is shown as a vertical bar. The file has parameters R and E; the number of bytes with timestamps in an interval of length T cannot exceed $TR + E$, as shown.

- Suppose a client plays several files, transmitted via network connections from different CMFS servers, in synchrony on a single workstation. The I/O server on the workstation handles the synchronization; it begins output only when sufficient data has been received on each connection (as done, for example, by the ACME server [3]). CMFS handles this case with no client intervention: logical clocks pause during the initial synchronization period, and resume thereafter.
- The client can, if the hardware is fast enough, read arbitrarily far ahead of the logical clock. This ‘‘workahead’’ data can then be buffered (in distributed applications, the buffers may be spread across many nodes), protecting against playback glitches and allowing improved system response to transient workload.

3. SYSTEM INTEGRATION ISSUES

3.1. Implementation Alternatives

Our implementation of CMFS runs as a user-level process on UNIX, accesses a SCSI disk via the UNIX raw disk interface, and communicates with clients via TCP connections. Other architectures, however, are possible (see Figure 4). For example, CMFS could be implemented at either kernel or user level. The client control interface could be provided either as system calls (for local clients only) or by remote procedure call.

The client data interface (the mechanism by which data is removed from a read session’s FIFO) can take several forms. A general approach is to have data sent out on a flow-controlled network connection, established as part of the `request_session()` call. Data is removed from the FIFO whenever the protocol allows it. If the client is a user-level process on the same machine as CMFS, the FIFO could be a shared-memory buffer or ‘‘memory-mapped stream’’ [6]. Finally, if the data is consumed by an I/O device on the same machine as CMFS, the FIFO might reside in kernel memory, accessed directly by the I/O device interrupt handler.

These alternatives have different performance implications. For example, a kernel-level implementation might have lower CPU scheduling overhead (most work could be done at the interrupt level) and improved control of physical memory. However, the issues addressed by CMFS (disk scheduling, buffer allocation, *etc.*) apply regardless of the alternative chosen.

3.2. End-to-End Scheduling

In typical applications, CM data is handled by many shared hardware ‘‘resources’’: disk, CPU, network, bus, memory, *etc.* To provide applications with deterministic end-to-end performance, we must integrate the scheduling of all these resources. CMFS is designed to serve as part of such a ‘‘meta-scheduling’’ scheme, the *CM-resource model* [2]. In this scheme, each resource can be reserved in ‘‘sessions’’ with fixed workload and delay bounds. The parameterization of workload and delay lets resources ‘‘work ahead’’ on real-time streams so they can response quickly to non-real-time workload.

A CMFS session’s ‘‘cushion’’ parameter $\bar{Y}$ allows the adjacent resource (typically the CPU) to have looser delay bounds. For example, suppose a client runs on the same host as CMFS (as in Figure 4c) and does CPU processing on the data before sending it to the next resource (say, a network). The client determines an upper bound on the CPU time per unit of data it requires, and reserves a session with the CPU resource. This session has a delay bound, say Y. It is easy to show the following:

Claim 3. *Suppose, in the above situation, that the client successfully creates a CMFS session with cushion Y, starts the session when it has processed Y amount of data (say, at time 0), and thereafter processes data without (voluntarily) pausing. Then at all times $t \geq 0$, the client has processed at least Rt bytes of data (R is the session data rate).*

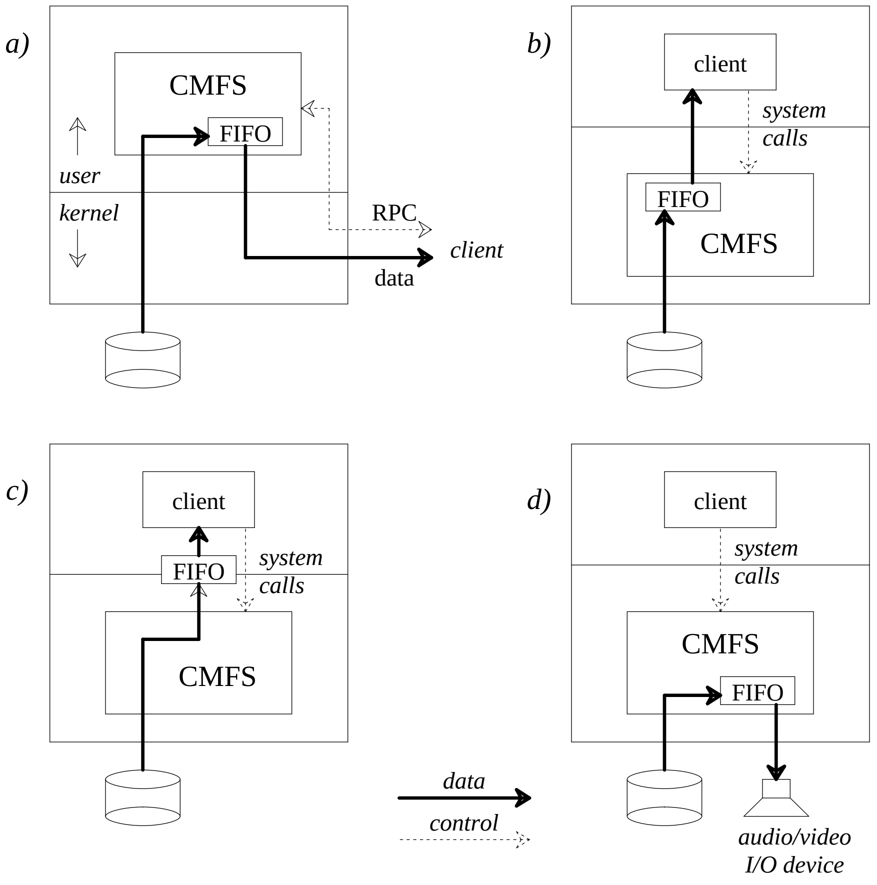


Figure 4: The CMFS prototype is a user-level process that communicates with its clients over network connections (a). Alternatively, it could be implemented in an OS kernel, with client data access by system calls (b), memory-mapped streams (c), or device interrupts (d).

(Note that this would *not* hold if the cushion were less than the CPU delay bound, because it would then be possible for the logical clock to stop.) Claim 3 can be generalized to bounded-rate files. This provides a more useful version of Claim 2, which makes the unrealistic assumption that client CPU processing is instantaneous and can be scheduled at precise instants.

4. DISK LAYOUT ASSUMPTIONS

We assume that the CMFS uses a single-spindle disk drive; disk operations are done sequentially. The disk is read and written in *blocks* of fixed size (a multiple of the hardware sector size). The CMFS reservation and scheduling algorithms do not mandate a particular disk layout. Instead, we assume that the layout allows the following ‘‘bounding functions’’ U_F and V_F to be obtained:

(1) For a given file F, $U_F(n)$ is an upper bound on the time to read n logically contiguous blocks of F (including all seek and rotation time), independent of the position of the disk

head and the starting block number to be read.

(2) $V_F(i, n)$ is an upper bound on the time needed to read the n blocks of file F starting at block i.

The bounds need not be tight; slackness in the bounds may, however, cause sessions to be rejected unnecessarily.

The functions U and V should take into account sector interleaving, interrupt-handling latency, the CPU time used by CMFS itself, features (such as track buffering) of the disk controller, and bad sectors detected when the disk is initialized.

4.1. Examples of Disk Layouts

Our CMFS prototype uses contiguous allocation. Each file begins at some point within a cylinder, filling the remainder of that cylinder, zero or more adjacent cylinders, and part of a final cylinder. The number of sectors per block is a fixed parameter. Ignoring CPU overhead and other factors, bounds functions for this layout are easy to derive. We assume L_{seek_min} and L_{seek_max} are bounds on the 1-track seek time and the worst-case seek time respectively, L_{block} is the time to read one block, $L_{rotation}$ is the rotation time, and N is the number of blocks per cylinder. Furthermore, we assume that the controller does track-buffering; it reads a track into a local buffer immediately after seeking to it. Hence, if an entire track is read, rotational latency is negligible regardless of the order in which the sectors are read. Possible bounds functions are then

$$U_F(n) = L_{seek\_max} + nL_{block} + \left\lceil \frac{n}{N} \right\rceil L_{seek\_min} + 2L_{rotation} \tag{10}$$

and

$$V_F(i, n) = L_{seek\_max} + nL_{block} + (k-1)L_{seek\_min} + \frac{j}{N} L_{rotation}$$

where k is the number of cylinders storing the n blocks of F starting at offset i, and j is the number of blocks not in F in the first and last of these cylinders. To account for CPU overhead, it would be necessary to leave a gap between blocks (perhaps by interleaving them) and to modify U and V accordingly.

A contiguous layout policy is feasible for read-only file systems or if disk space is abundant. For more flexibility, a variant of the 4.2BSD UNIX file system layout [8] could be used. A real-time file might consist of clusters of n contiguous blocks, with every sequence of k clusters constrained to a single cylinder group. n and k are per-file parameters; they are related to the `max_rate` parameter of the file. Bounds functions U and V can be computed from n, k, the size of a cylinder group, the disk parameters, and the overhead of reading and writing allocation map and index blocks. Allocation and compaction strategies would pose a complex set of issues; we do not discuss them here.

5. ACCEPTANCE TEST

CMFS can accept a new session S only if its data rate requirements, together with those of existing sessions, can be guaranteed. For this, a sufficient condition is the existence of a static schedule (that cyclically reads fixed numbers of blocks of each session) satisfying the rate requirements of all session under worst-case assumptions, and for which enough buffer space is available.

In this section we give an algorithm for deciding whether such a schedule exists. The algorithm constructs the shortest such schedule; this minimal static schedule also plays an important role in dynamic scheduling (see Section 6).

5.1. Properties of Static Schedules

Suppose that sessions $S_1 \cdots S_n$ read files $F_1 \cdots F_n$ at rates $R_1 \cdots R_n$. An *operation set* ϕ assigns to each S_i a positive integer M_i. CMFS *performs* an operation set by seeking to the next block of file F_i, reading M_i blocks of the file, and doing this for every session S_i (the order of operations is not specified). From Section 4,

$$L(i) = U_{F_i}(M_i)$$

$$L(\phi) = \sum_{i=1}^{n} U_{F_i}(M_i) \tag{11}$$

are upper bounds on the elapsed time of S_i's operation and ϕ as a whole, respectively.

The data read in ϕ ‘‘sustains’’ session i for a period $\frac{M_i A}{R_i}$, where A is the block size in bytes; we denote this period $D_i(\phi)$. $D(\phi)$, the period for which the data read in ϕ sustains all the sessions, is then $\min_{1 \le i \le n} D_i(\phi)$.

If the data read in ϕ ‘‘lasts longer’’ than the worst-case time it takes to perform ϕ, we call it a *workahead-augmenting set* (WAS). This holds if $L(\phi) < D(\phi)$, or equivalently

$$M_i A > R_i L(\phi) \tag{12}$$

for all i.

If the amount of data read for each session in an operation set ϕ fits in the corresponding FIFO, we say that ϕ is *feasible*. This holds if, for all i,

$$(M_i + 1)A + \bar{Y}_i \le B_i \tag{13}$$

where B_i is the size of the FIFO buffer used by S_i and $\bar{Y}_i$ is the cushion parameter of S_i (Section 2.1). This inequality reflects the worst case in which a fractional block is already buffered and the client has used none of its cushion.

An *operation sequence* Φ is a pair (π, ϕ), where π is a permutation of $1 \cdots n$ and ϕ is an operation set. CMFS *performs* an operation sequence by doing the operations in ϕ in the order $\pi(1) \cdots \pi(n)$. Φ is called *workahead-augmenting* if ϕ is workahead-augmenting; likewise Φ is *feasible* if ϕ is feasible.

At time t, the duration of data buffered for session S_i, above and beyond the client's cushion $\bar{Y}$, is given by $(P(t) - C(t) - \bar{Y})/R$, where P, C, $\bar{Y}$ and R are the parameters for S_i. We call this the ‘‘workahead’’ of S_i and denote it by $W_i(t)$. We say that S_i *starves* if W_i becomes negative. The *file system state* $W(t)$ at time t is the vector $<W_1(t) \cdots W_n(t)>$.

Let Φ be an operation sequence. Suppose that enough data is buffered so that, if Φ is performed immediately, no session starves before its operation is completed. We then say that the state is *safe relative to* Φ. This holds if

$$W_{\pi(j)}(t) \ge \sum_{i=1}^{j} L(\pi(i))$$

for all j (recall that $L(j)$ is the worst-case time needed for S_j's operation).

The following two claims show that CMFS can accept a set of session if there is a feasible workahead-augmenting sequence. The order π is not important; for simplicity we assume it is the identity permutation.

Claim 4. *If Φ is workahead-augmenting and feasible, then there is a system state that is safe relative to Φ.*

Proof. Let W be the state in which all buffers are full. Then for all j we have

$$\begin{aligned} W_j(t) &= (P(t) - C(t) - \bar{Y})/R \\ &= (B_j - \bar{Y})/R \\ &> (M_j A)/R \\ &> L(\Phi) \\ &\geq \sum_{i=1}^{j} L(i) \end{aligned}$$

so W is safe relative to Φ (the final three steps use Eqs. 13, 12, and 11). □

Claim 5. *Suppose that there is a feasible workahead-augmenting operation sequence Φ, and assume that at time t_0 the state W is safe relative to Φ. Then the CMFS can satisfy the Read Session Axioms (Eqs. 1 and 2) for all i and $t \geq t_0$.*

Proof. We prove this by defining a disk scheduling policy, called the **Static** policy, that satisfies the axioms. The policy is as follows. Repeatedly apply the schedule given by Φ, with the following exception: if $P(t) + A - G(t) > B_i$ at the point of starting a block read for S_i, then immediately skip to the next session (since reading the block could cause a buffer overflow). It is clear that this policy preserves Eq. 1.

Consider a particular session S_k during one ‘‘cycle’’ of Φ, starting at time $t = 0$ (see Figure 5). Let t_D denote the time at which S_k’s operation ends. Eq. 2 holds during $[0, t_D]$ since

$$t_D \leq \sum_{i=1}^{k} L(i) \leq W_k(0)$$

and $C(t)$ advances by at most R bytes/second. The operation for S_k either reads the full amount or is truncated; in either case

$$W_k(t_D) > L(\Phi) \tag{14}$$

since Φ is workahead-augmenting. Let t_E denote the time at which the cycle ends. Then

$$t_E - t_D \leq \sum_{i=k+1}^{n} L(i) \tag{15}$$

Combining Eq. 14, Eq. 15, and the clock rate bound R, we see that W_k remains positive during $[t_D, t_E]$, and moreover

$$W_k(t_E) \geq \sum_{i=1}^{k} L(i) \tag{16}$$

Therefore no starvation occurs during the cycle, and (by Eq. 16) the state W at the end of the cycle remains safe relative to Φ. Hence the Static scheduling policy maintains the Read Session Axioms for all sessions. □

5.2. The Minimal Feasible WAS

Claim 5 shows that CMFS can satisfy the data rates of a set of sessions if there is a feasible WAS. We now describe an algorithm to compute the *minimal feasible WAS* $\bar{\phi}$ (the feasible WAS for which $L(\phi)$ is least). Clearly, a minimal feasible WAS exists if and only if a feasible WAS exists.

Suppose that sessions $S_1 \cdots S_n$ are given. Let D_i be the ‘‘duration’’ of one block of data for S_i, given by A/R_i. Let $\{t_0 < t_1 < \cdots\}$ be the set of numbers of the form kD_i for $k \geq 0$ and $i \geq 0$

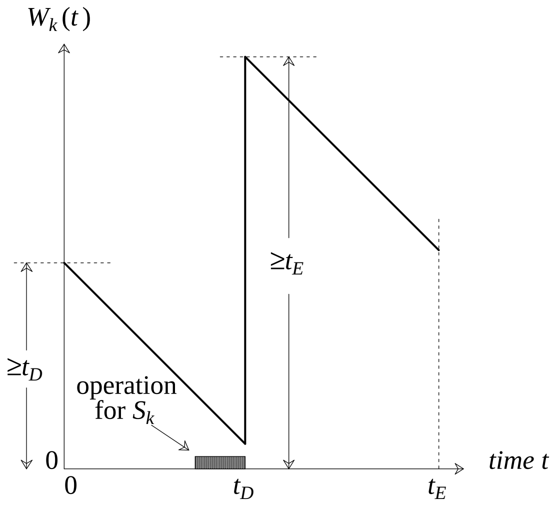


Figure 5: Diagram for the proof of Claim 5, showing the workahead W_k of a session S_k during one cycle of the Static scheduling policy. At the start of the cycle, S_k has enough workahead to last until time t_D, when its operation is finished. The amount of data read suffices for at least $L(\phi)$, which exceeds the length t_E of the cycle. Therefore W_k is always positive; *i.e.*, S_k never ‘‘starves’’.

(see Figure 6). Let I_i denote the interval $(t_i, t_{i+1}]$. Let ϕ_i denote the operation set $< \lceil R_1 t_i \rceil \cdots \lceil R_n t_i \rceil >$. Note that ϕ_{i+1} differs from ϕ_i by the addition of 1 block to all sessions whose data periods divide t_{i+1}; hence the sequence of ϕ_i is easy to compute. Note also that $L(\phi_i) < L(\phi_{i+1})$ for all i.

Claim 6. *If ϕ is an operation set such that $D(\phi) \in I_i$, then $L(\phi) \geq L(\phi_i)$.*

Proof. Any sequence ϕ for which $D(\phi) > t_i$ must read at least $\lceil t_i R_j \rceil$ blocks for each session j, and hence $L(\phi) \geq L(\phi_i)$. □

Claim 7. *If there is a feasible WAS ϕ such that $D(\phi) \in I_i$, then ϕ_i is feasible.*

Proof. ϕ must read at least as many blocks for each session as does ϕ_i. Therefore, since ϕ is feasible, so is ϕ_i. □

Claim 8. *The following algorithm computes the minimal WAS:*

(1) Let $\phi_0 = <1, \cdots, 1>$ (this is the minimal operation set for which $D(\phi) \in I_0$).

(2) If ϕ_i is infeasible (*i.e.*, there is no allocation $< B_1 \cdots B_n >$ of buffer space to client FIFOs such that $M_i A + \bar{Y}_i \leq B_i$ for all i) stop; there is no feasible WAS.

(3) If $L(\phi_i) \leq D(\phi_i)$ stop; ϕ_i is the minimal feasible WAS.

(4) Compute ϕ_{i+1} and go to (2).

Proof. Suppose the algorithm stops in step 3, returning a WAS in I_j. Let ϕ be the minimal WAS, and let i be such that $D(\phi) \in I_i$. It is not possible that $i < j$, since then ϕ_i is feasible (Claim 7) and workahead-augmenting, so the algorithm would have terminated at iteration i. It is also

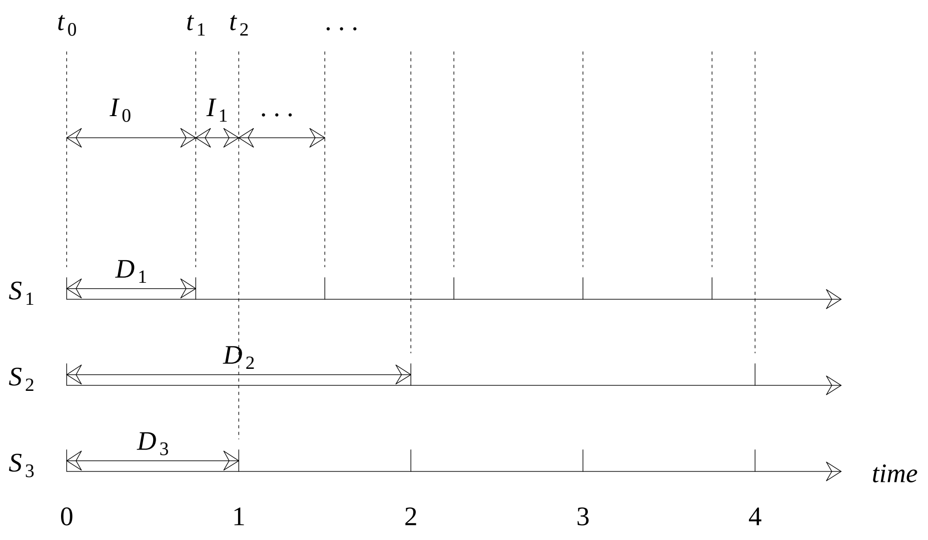


Figure 6: A block of data for session S_i has a ‘‘duration’’ D_i that depends on the data rate of S_i. The set of all multiples of these periods defines a set of intervals I_i. Within each interval there is a unique minimal-length operation set ϕ_i. By enumerating the ϕ_i we can find the minimal feasible WAS.

not possible that $i>j$, since then $L(\phi) \geq L(\phi_i) > L(\phi_j)$, contradicting the minimality of $L(\phi)$. Therefore $i=j$ and (from Claim 6) we must have $L(\phi_i) = L(\phi)$, so ϕ_i is the minimal WAS.

Finally, suppose that the algorithm terminates in step 2 for some i. Suppose that a feasible WAS ϕ exists, with $D(\phi) \in I_j$. By the above arguments $i \leq j$. But then ϕ reads at least as many blocks for each session as does ϕ_i, so the buffer allocation feasible for ϕ is feasible for ϕ_i, which is a contradiction. □

5.3. Buffer Space Allotment

Suppose that a fixed amount B of buffer space is available for CMFS client FIFOs. How should this space be divided among the various clients? CMFS performs best when all sessions can ‘‘work ahead’’ by about the same time (see Section 7). In other words, the buffer space allocated to a session, beyond that needed for the client’s cushion $\bar{Y}$, should be roughly proportional to the data rate R.

CMFS therefore uses following policy. Let $Y = \sum_{i=1}^{n} \bar{Y}_i$ and $R = \sum_{i=1}^{n} R_i$. Session S_j is allocated

$$\bar{Y}_j + \frac{(B-Y)R_j}{R} \tag{17}$$

bytes. This allocation is rounded up, if needed, to a multiple of the memory-allocation block size.

6. DISK SCHEDULING POLICY

On completion of each disk block I/O, CMFS decides which disk block to read or write next, and issues the appropriate command (seek, read, or write) to the disk device driver. The algorithm for this decision constitutes a *disk scheduling policy*. Such a policy must prevent starvation of current sessions, and must delay the return of the `request_session()` call for a newly accepted session until it is safe to do so. It should also handle non-real-time workload efficiently. Policies for real-time CPU scheduling, such as earliest-deadline-first [7], are not immediately relevant because of seeks. In this section we describe several possible disk scheduling policies. Some of these policies are defined in terms of *slack time,* which we will now define.

6.1. Slack Time

If, at a particular time, enough data is buffered for all sessions, CMFS is free to do non-real-time operations or workahead for real-time sessions. The amount of this ‘‘slack time’’, denoted H, is computed as follows. Suppose that the minimal WAS $\bar{\phi}$ takes worst-case time $L(1) + \cdots + L(n)$, where $L(i) = U_{F_i}(M_i)$. Let π be a permutation of $1 \cdots n$, and let Φ be the operation sequence $(\pi, \bar{\phi})$. If Φ is performed immediately, the workahead of session j will not fall below $H_j = W_j - \sum_{i=1}^{j} L(\pi(i))$ (H_j is called the slack time of session j). CMFS can safely defer starting Φ for a period of $H = \min_{i=1}^{n}(H_i)$.

Claim 9. *Let $\bar{\pi}$ be the ordering of sessions by increasing value of workahead W_i. Then, among all permutations π, $\bar{\pi}$ gives the maximal value of H.*

Proof. Consider a permutation π in which W_i is not increasing; in particular suppose $W_i > W_j$, where $\pi(i) + 1 = \pi(j)$. Let B and C denote the slack times of the two sessions. If we reverse the order of the two sessions in π, then for the new slack times D and E we have $C < D$ and $C < E$ (see Figure 7). The slack times of other sessions remain unchanged. Hence the minimum of the slack times is not decreased by reversing the order. □

We therefore consider only the increasing-workahead ordering $\bar{\pi}$ of sessions. Let $\bar{\Phi}$ denote $(\bar{\pi}, \bar{\phi})$, and let H_i and H denote the corresponding slack times at a particular moment. It is important to note that $H < 0$ does not imply that starvation has occurred or will occur. H is based on the pessimistic assumption that an inter-file seek is needed prior to every operation in the WAS. For example, if $H < 0$ during of a multi-block file operation, CMFS is compelled to finish the current operation; starting a new WAS would incur an inter-file seek.

6.2. Real-Time Scheduling Policies

We now describe several possible disk scheduling policies. These policies all avoid starvation; their relative performance is discussed in Section 7.

(1) **The Static/Minimal policy** (a special case of the Static policy described in Section 5) simply repeats the minimal WAS.

(2) The **Greedy policy** does the longest possible read for each session. At each iteration, it computes the slack time H, finds the session S_i with smallest workahead, and reads blocks for S_i for a period of $H + L(i)$; in other words, it devotes the entire slack time to reading ahead on S_i.

(3) The **Cyclical Plan policy** differs from Greedy in that it tries to distribute current slack time among the sessions in a way that maximizes future slack time. It augments the minimal WAS $\bar{\Phi}$ with H seconds of additional reads (these reads are done, for each session S_i, immediately after the read for S_i in $\bar{\Phi}$). The policy distributes workahead by identifying the ‘‘bottleneck session’’ (that for which H_i is smallest) and schedules an extra block for it, updating H_i and H; this is repeated until H is exhausted. The resulting

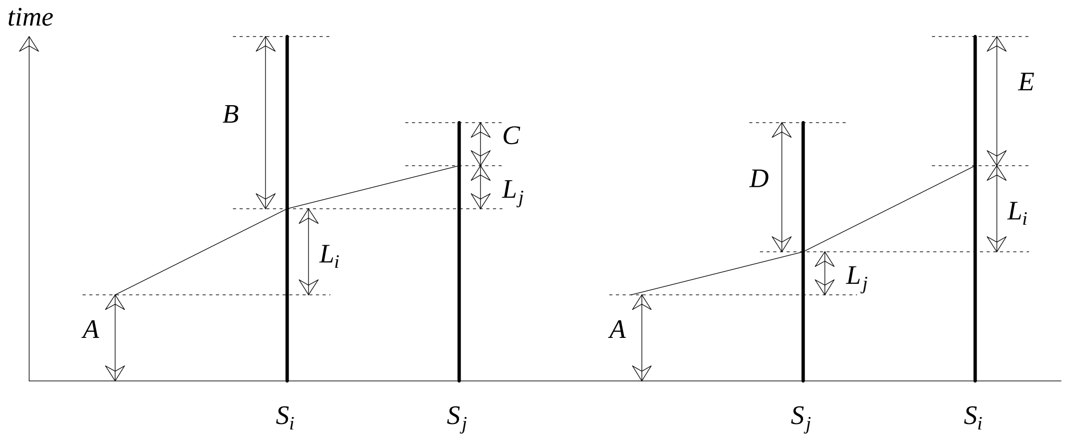


Figure 7: Slack time is maximized by ordering sessions by increasing workahead. Suppose a sequence has two sessions S_i and S_j that are not in this order. The bold lines represent their workaheads W, and their slack times are B and C as shown (A denotes the maximum time needed for operations preceding i; L_i is the time bound for the operation of session S_i). By reversing the order of the two sessions, the slack times are D and E. Simple algebra shows that $C < D$ and $C < E$. Global slack time is the minimum of the session slack times, so the result follows.

schedule determines the number for blocks read for the least-workahead session; when this read completes, the procedure is repeated.

In both the Greedy and Cyclical Plan policies, the least-workahead session is serviced immediately. Therefore the value of H used by these policies can be computed as the minimum of the slack times of all sessions *except* the least-workahead session, yielding **Aggressive** versions of each policy. All policies skip to the next session when a buffer size limit is reached. If at some point all buffers are full, no operation is done; when a client subsequently removes sufficient data from a FIFO, the policy is restarted.

Greedy and Static/Minimal have low CPU overhead: in our prototype, on a 15 MIPS workstation and with three sessions, they use about 200 microseconds per scheduling decision. Because Cyclical Plan builds its schedule one block at a time, it uses CPU time proportional to buffer space on each transition between sessions. This limits its utility.

6.3. Non-Real-Time Operations

A non-real-time operation N with worst-case latency L can safely be started if $L \le H$. However, the policy of servicing non-real-time operations whenever it is safe to do so may tend to keep H low. This forces the scheduler to do short real-time operations (close to the minimal WAS), causing the system to run inefficiently. It may be preferable to do non-real-time operations only when H exceeds some nonzero threshold.

To avoid the seek overhead of rapidly alternating between real-time and non-real-time operations, CMFS uses the following *slack time hysteresis* policy for non-real-time workload. An interactive operation can be started whenever $H \ge H2$. Initially, interactive operations can be started if $H \ge H_1$; however, if H falls below H_{I1} no further interactive operations are started until

H exceeds H_{I2}. Similarly, background operations are done within a hysteresis interval $[H_{B1}, H_{B2}]$. No background operation is started if an interactive operation is eligible to start. In Section 7 we examine the effects of hysteresis, and of the hysteresis parameters, on system performance.

6.4. Session Startup

A newly-accepted session is said to *start* when its `request_session()` call returns. This must occur only when the system state is safe with respect to the new WAS. A special mechanism is needed for handling this ‘‘startup’’ phase.

Suppose sessions $S_1 \cdots S_n$ are currently active, and session S_{n+1} has been accepted but not yet started. Let ϕ_n and ϕ_{n+1} denote the feasible WASs for the sets $S_1 \cdots S_n$ and $S_1 \cdots S_{n+1}$ respectively. S_{n+1} is started as follows. CMFS adjusts FIFO buffer sizes according to the procedure described in Section 5.3. It can shrink a buffer by discarding data from the end of the FIFO if needed (it must later reread the data from disk). The scheduler then goes into ‘‘startup mode’’ during which its policies are changed as follows:

(1) Non-real-time operations are queued for later execution.

(2) For scheduling purposes, slack time H is computed relative to ϕ_n. However, in the Cyclical Plan policy the allocation of slack time for workahead is done relative to ϕ_{n+1}, using a session ordering in which the new session appears first (however, no I/O for S_{n+1} is done during this phase).

(3) When the system state is safe with respect to ϕ_{n+1}, a read of $\phi_{n+1}(n+1)$ blocks for S_{n+1} is started. When this read is completed, the system state is ‘‘safe’’ for all $n+1$ sessions. The `request_session()` call for S_{n+1} is allowed to return, ϕ_{n+1} becomes the system’s WAS, and the system leaves startup mode.

Step (3) can be omitted for write sessions because the equivalent read session starts with a full buffer (Section 2.3).

7. PERFORMANCE

In this section we study the effects of disk scheduling policies and hardware parameters on CMFS performance. Our study uses simulation. We chose not to use the ‘‘real I/O’’ version of CMFS because of the scheduling vagaries of UNIX, the poor performance of its SCSI disk I/O, and the restriction to our available disk.

We wrote the CMFS prototype so that disk I/O operations can optionally be simulated rather than performed. The simulator keeps track of the disk head radial and rotational position, and models latencies realistically. Other actions (*e.g.*, CPU execution) are modeled as instantaneous. Unless otherwise stated, the simulations use the Cyclical Plan policy, and assume a disk with 11.8 Mbps transfer rate and 39 ms worst case seek time. Block size is 512 bytes.

7.1. Number of Concurrent Sessions

Figure 8 shows the maximum number of concurrent sessions accepted by CMFS as a function of total buffer space. This is shown for two different session data rates: 64 Kbps and 1.4 Mbps. In each graph, curves are given for three different disk types: 39 ms maximum seek time and 11.8 Mbps transfer rate (CDC Wren V), 35 ms maximum seek time and 8.6 Mbps transfer rate (CDC Wren III) and, 180 ms maximum seek time and 5.2 Mbps transfer rate (Sony 5.25" optical disk).

The disk transfer rate imposes an upper bound on the number of concurrent sessions that can be accepted. Unbounded buffer space is needed as this limit is approached. To reach 90% of the limit with a Wren V disk requires 4 MB for 1.4 Mbps sessions and 85 MB for 64 Kbps sessions. The efficiency depends on the length of operations in the minimal WAS; 64 Kbps sessions require a proportionally longer WAS and therefore more buffer space. When the number of accepted sessions is fixed, a disk with higher seek time needs a longer minimal WAS and therefore more buffer space.

7.2. Performance of Disk Scheduling Policies

Since non-real-time operations can be done only if there is enough slack time, an important criterion for disk scheduling policies is how quickly they increase slack time. To study this, we simulated CMFS with three concurrent 1.4 Mbps sessions, no non-real-time traffic, and 8 MB system buffer size. From the results (Figure 9) we see that

Cyclical Plan performs slightly better than Greedy when slack is low, but Greedy quickly catches up. Static/Minimal, because it cannot do long operations, performs much worse at higher slack levels. With appropriate hysteresis values CMFS maintains moderate slack levels during steady-state operation; thus the dynamic policies are preferable.

7.3. Response Time of Interactive Traffic

To study the effect of real-time traffic on interactive traffic, we simulated a fixed number of sessions together with interactive requests that read randomly positioned blocks from disk. The interactive request arrival is Poisson with mean arrival rate λ. We define the *response time* of an interactive request as the time from its arrival to the start of the disk operation; the delay of the operation itself, including the seek, is not included.

The effect of hysteresis parameters is most noticeable under heavy load (otherwise slack remains high and hysteresis is not exercised). Figure 10 plots mean interactive response time

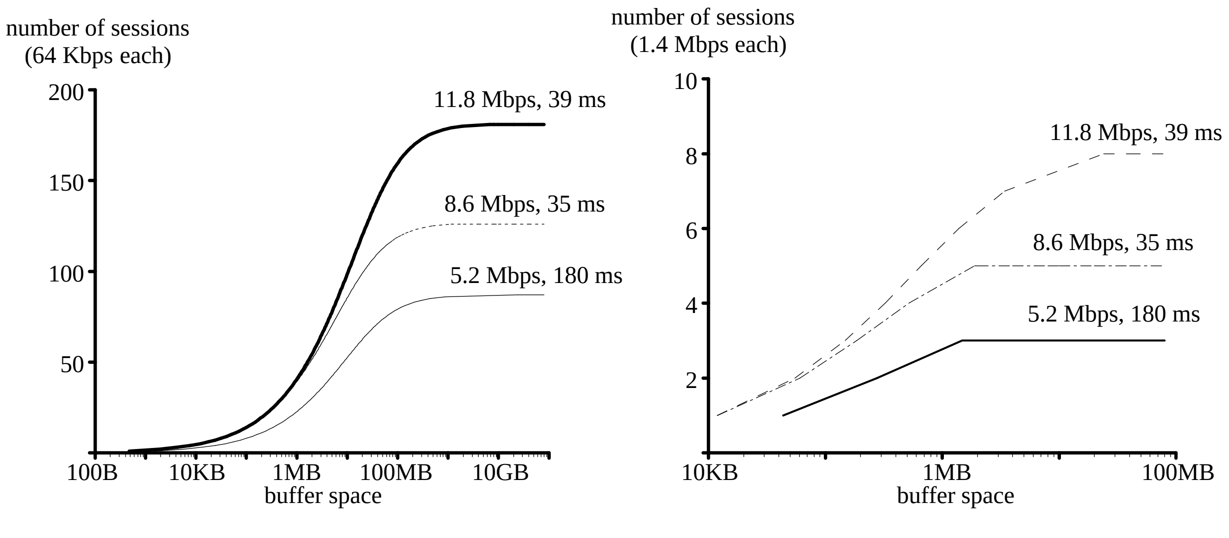


Figure 8: The number of sessions that can be accepted by CMFS depends on the available buffer space. The disk transfer rate imposes an upper limit on the number of sessions; to reach 90% of the limit with the 11.8 Mbps disk requires 4 MB of buffer space for 1.4 Mbps sessions and 85 MB for 64 Kbps sessions.

Figure 9: Disk scheduling policies build up slack at different rates. The Aggressive Cyclical Plan (solid line), Aggressive Greedy (dotted) and Static/Minimal (dashed) policies are shown here (the non-aggressive versions, not shown, performed slightly worse).

against H_{I2}, for different values of H_{I1}, under a heavy load. We observe that:

- For fixed H_{I1} and increasing H_{I2}, interactive response time drops steeply and then rises gradually. When H_{I1} nearly equals H_{I2}, interactive response is poor because the system switches rapidly between real-time and non-real-time operations, causing high seek overhead. As H_{I2} increases, this oscillation becomes less frequent and response improves. Since interactive requests are queued while slack builds up from H_{I1} to H_{I2}, response degrades if H_{I2} is increased past a certain point. For all H_{I1}, response is best when $H_{I2} - H_{I1}$ is about 0.5 seconds.
- When H_{I1} is very small (*e.g.*, 0.1 sec in Figure 10), slack builds up slowly from H_{I1} to H_{I2} so response is poor. A similar effect is observed if H_{I2} exceeds about $0.95Hmax$ ($H_{\max}$ denotes the upper bound on H imposed by buffer space), since it is difficult to fill all buffers simultaneously.

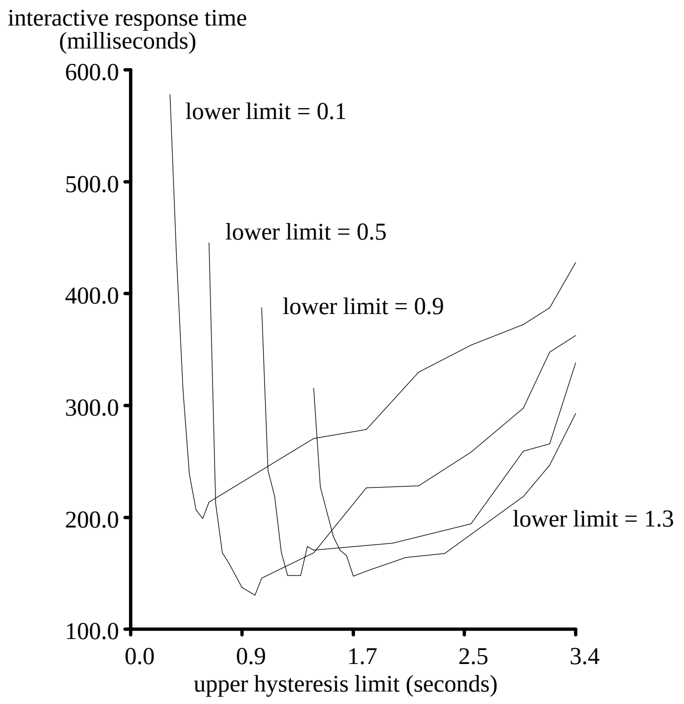


Figure 10: The effect of the hysteresis limits H_{I1} and H_{I2} on the mean response time for non-real-time requests of the interactive class. This experiment was conducted with 2 MB total buffer ($H_{\max}$ is 3.4 seconds), 20 interactive arrivals per second and three 1.4 Mbps sessions.

Based on these observations, reasonable ‘‘rule of thumb’’ values are $H_{I1} = H_{\max}/3$ and $H_{I2} - H_{I1} = \min(H_{\max}/3, 0.5)$.

Figure 11 plots mean interactive response time as a function of arrival rate, for different values of system buffer size. If interactive arrival rate is low, system slack stays near $H_{\max}$ (Figure 12) and most interactive requests are serviced without waiting for real-time traffic. At higher arrival rates (in Figure 11, about 5 per second for the 500 KB case and 10 per second for 1 MB), interactive response degrades because slack sometimes reaches the lower hysteresis limit H_{I1}, and interactive requests then are blocked until slack reaches H_{I2}.

7.4. Throughput of Background Traffic

To estimate the effect of real-time traffic on background traffic throughput, we simulated three 1.4 Mbps sessions and a single background task that sequentially reads a long, contiguously-allocated file. We define the *background throughput fraction* T as the fraction of residual disk bandwidth (*i.e.*, disk bandwidth not taken up by real-time sessions) used by the background task. For the same reasons as discussed in Section 7.3, T is low if H_{B1} is very small, H_{B2} is close to $H_{\max}$, or $H_{B2} - H_{B1}$ is small. In this case (since throughput, rather than response time, is the goal) there is no penalty if $H_{B2} - H_{B1}$ is large. We found that T was maximized for (roughly) $H_{B1} = H_{\max}/4$ and $H_{B2} = .9Hmax$. Figure 13 plots T (with these hysteresis limits) against buffer space.

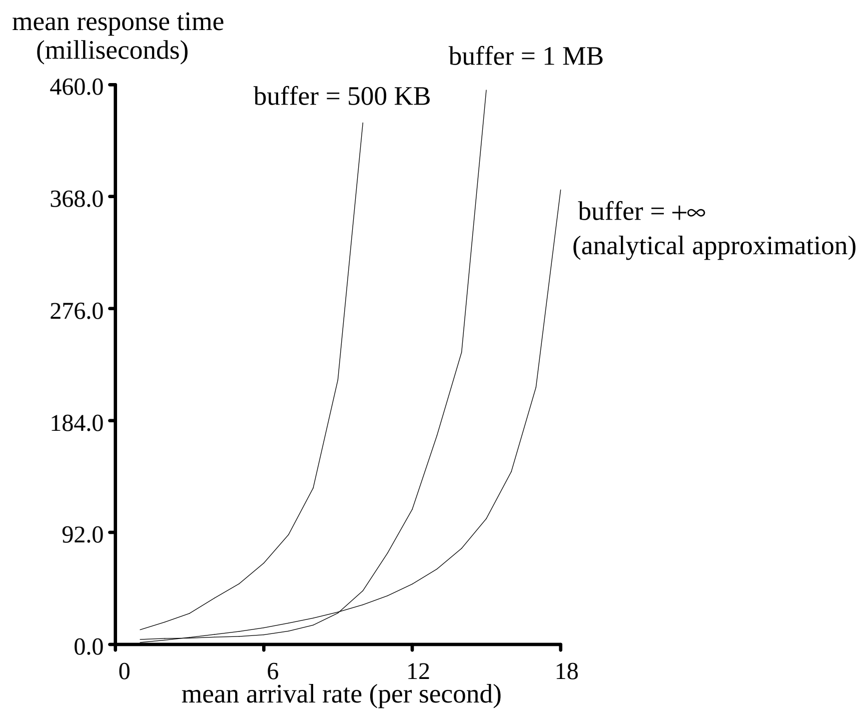


Figure 11: Mean interactive response time as a function of arrival rate, for different values of total buffer space. In this experiment there were four concurrent 1.4 Mbps sessions. The hysteresis limits (0.04, 0.08) and (0.35, 0.65) were used for the 500 KB and 1 MB cases respectively. The ‘‘infinite buffer’’ curve was obtained analytically, modeling the file system as an M/G/1 queue.

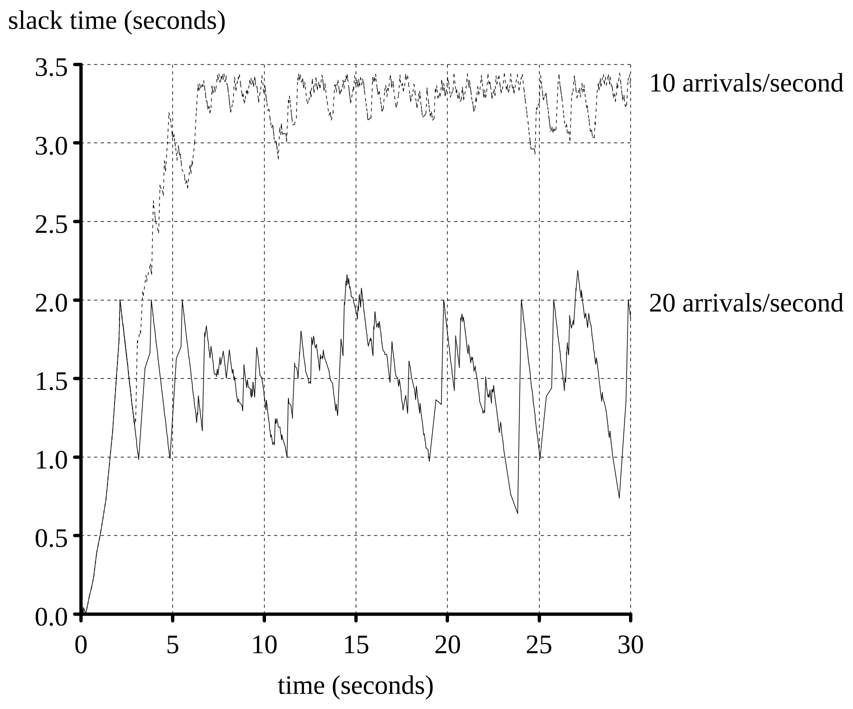


Figure 12: The variation of slack time H in the presence of interactive non-real-time traffic. Three 1.4 Mbps sessions start at time zero, the system has 2MB of buffer space ($H_{\max}$ is 3.4 seconds), and hysteresis limits are H_{I1} = 1.0 and H_{I2} = 2.0. If the interactive arrival rate is low (10 per second in this case), H stays near $H_{\max}$. For high arrival rates (20 per second), H oscillates between the hysteresis limits.

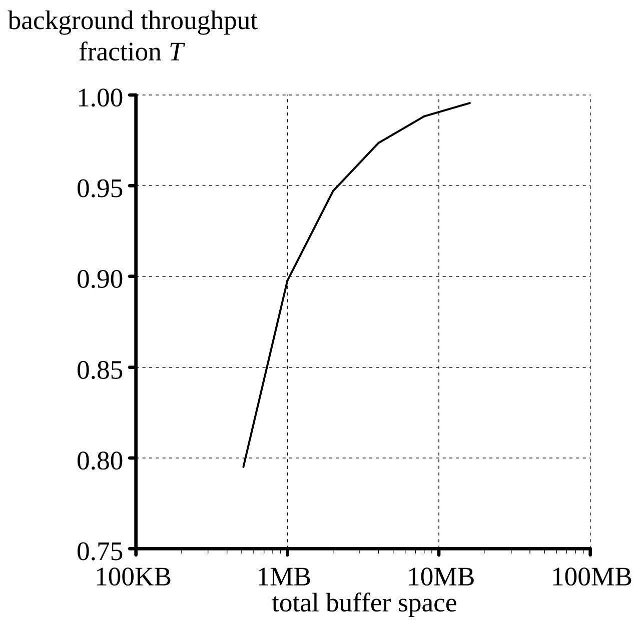


Figure 13: Background throughput as a function of total buffer space, with a real-time workload of three 1.4 Mbps sessions. As buffer space increases so does $H_{B2} - H_{B1}$; this allows longer periods of background I/O, and hence less seek overhead.

7.5. Session Startup Time

Write sessions can usually be started immediately; see Section 2.3. To study startup time for read sessions, we ran a simulation in which requests for six sessions arrive at time zero. Figure 14 shows the start times of the sessions. The difference between successive start times increase; this is because the workaheads of existing sessions have to be increased to accommodate the new minimal WAS, which becomes longer as more sessions are added. Startup times are on the order of one second, which is similar to the startup time of a consumer VCR. However, it is too large for applications that require instantaneous response, such as interactive musical performance using sounds stored on disk. This problem can be solved by storing an initial segment of each sound file in memory.

8. RELATED WORK

Structural issues for multi-media files (sharing, parallel composition, annotations, *etc.*) have been addressed in the Xerox Etherphone system [14], the Sun Multimedia File System [13], and the Northwestern Network Sound System [12]. These projects do not concentrate on performance or scheduling issues, and the systems cannot make performance guarantees.

Other projects have addressed performance but without hard guarantees. Abbott gives a qualitative discussion of disk scheduling for playback of multiple audio tracks [1]. He compares a ‘‘balanced’’ policy in which read-ahead is divided among sessions, to a shortest-seek-first policy. His analysis does not, however, provide an acceptance test or performance guarantees.

Park and English [9] describe a system supporting single channel audio playback. Non-real-time traffic may concurrently access the disk, causing available disk bandwidth to change.

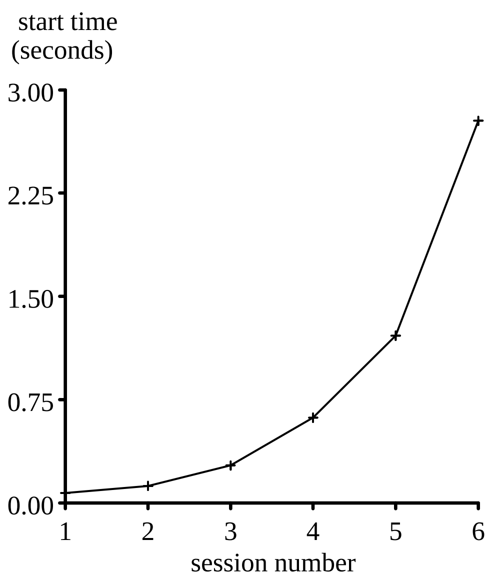


Figure 14: When six 1.4 Mpbs session requests arrive simultaneously at time zero, their actual start times are staggered as shown.

As an alternative to disk bandwidth reservation for the audio channel, they propose changing the data rate of the channel dynamically, to accommodate non-real-time workload. The high data rate is chosen if the workahead on the stream is above a fixed threshold. This strategy does not guarantee a minimum data rate.

Yu *et al.* [15] discuss the layout of interleaved data streams with different data rates on a compact disk for guaranteed-performance playback. Their assumptions (single session, fixed rates, small buffers, no non-real-time traffic) are more restrictive than ours.

Gemmell and Christodoulakis [5] describe a file system supporting multiple audio channel playback with concurrent non-real-time traffic. Like CMFS, this work provides a basis for hard performance guarantees. However, it differs from CMFS in several respects. The channels must have the same (constant) data rate and must start at the same time. The scheduling policy is static: the system repeatedly applies a single feasible WAS for the audio channels, and reserves ‘‘free’’ time during each operation sequence to service non-real-time traffic. For non-real-time traffic, this static policy may perform worse than CMFS because 1) CMFS can ‘‘interrupt’’ a WAS, allowing non-real-time traffic to start immediately, and 2) CMFS can use accumulated system slack to handle long bursts of non-real-time traffic.

Rangan and Vin [11] describe a system that combines disk input and display-device output for multiple data streams. They give expressions for admission control under the assumption that streams have equal data rates. Their disk scheduling policy is similar to Static/Minimum.

9. CONCLUSION

The Continuous Media File System (CMFS) provides guaranteed-performance read and write ‘‘sessions’’. Several such sessions can coexist with non-real-time workload on a single disk. The central ideas of CMFS include the following:

- *Semantics:* The CMFS session interface supports a range of client requirements, including variable-rate data, starting and stopping, synchronization of multiple streams, and client workahead. The semantics are defined rigorously (Section 2.1), but they include a ‘‘cushion’’ factor $\bar{Y}$ that provides flexibility in client CPU scheduling.
- *Layout:* CMFS requires that bounds functions U and V can be obtained (Section 4), but does not mandate a particular disk layout.
- *Session acceptance:* To decide if a session request can be accepted, CMFS checks if a *feasible WAS* (Section 5.2) exists.
- *Disk scheduling:* We found that dynamic policies (Greedy and Cyclical Plan) performed better than the Static/Minimal policy.
- *Concurrent non-real-time access:* CMFS handles non-real-time as well as real-time file access; disk space can be dynamically used for either purpose. CMFS uses the *slack time hysteresis* policy for scheduling non-real-time access. With appropriate parameters, this policy allows long non-real-time operations to complete without interruption.

9.1. Refinements and Future Work

The following observations suggest possible improvements to CMFS. First, for a session S_i, the graph of workahead W_i as a function of time is roughly a ‘‘sawtooth’’ function. If we consider two sessions S_1 and S_2 that have opposite phases in the scheduling cycle, then $\max(W_1 + W_2)$ is generally less than $\max(W_1) + \max(W_2)$. S_1 and S_2 can therefore share buffer space, possibly improving non-real-time performance or increasing the number of sessions that can be accepted. Second, the scheduling policy could take disk head position into account in various ways. For example, it could yield a session ordering that is more efficient than smallest-workahead-first (Section 6.1). Similarly, the use of a policy such as SCAN [4] for ordering non-real-time operations could improve their performance.

Although we have presented the CMFS algorithms in the context of a single-spindle disk drive, they are equally applicable to a disk array in which files are ‘‘striped’’ across multiple disks [10]. A client-level session could be composed of sessions on multiple disks, with each disk reserved and scheduled as described here. This could be used to provide sessions with data rates higher than those of the underlying disk drives. It could improve load-balancing and availability even for sessions with data rates lower than individual disks.

ACKNOWLEDGEMENTS

Discussions with Vassilios Polimenis, George Homsy, and Mark Moran contributed to the CMFS design.

REFERENCES

[1] C. Abbott, ‘‘Efficient Editing of Digital Sound on Disk’’, *J. Audio Eng. Soc. 32*, 6 (June 1984), 394.

[2] D. P. Anderson, ‘‘Meta-Scheduling for Distributed Continuous Media’’, UC Berkeley, EECS Dept., Technical Report No. UCB/CSD 90/599, Oct. 1990.

[3] D. P. Anderson and G. Homsy, ‘‘A Continuous Media I/O Server and its Synchronization Mechanism’’, *IEEE Computer*, Oct. 1991, 51-57.

[4] P. J. Denning, ‘‘Effects of Scheduling on File Memory Operations’’, *Proceedings of the AFIPS National Computer Conf. Proc. Spring Joint Computer Conference*, 1967, 9-21.

[5] J. Gemmell and S. Christodoulakis, ‘‘Principles of Delay-Sensitive Multimedia Data Storage and Retrieval’’, *ACM TOIS 10*, 1 (Jan. 1992), 51-90.

[6] R. Govindan and D. P. Anderson, ‘‘Scheduling and IPC Mechanisms for Continuous Media’’, *Proc. of the 13th ACM Symp. on Operating System Prin.*, Pacific Grove, California, Oct. 14-16, 1991, 68-80.

[7] C. L. Liu and J. W. Layland, ‘‘Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment’’, *J. ACM 20*, 1 (1973), 47-61.

[8] M. K. McKusick, W. N. Joy, S. J. Leffler and R. S. Fabry, ‘‘A Fast File System for UNIX’’, *ACM Transactions on Computer Systems 2,* 3 (Aug. 1984), 181-197.

[9] A. Park and P. English, ‘‘A Variable Rate Strategy for Retrieving Audio Data From Secondary Storage’’, *Proceedings of the International Conference on Multimedia Information Systems*, Singapore, Jan. 1991, 135-146.

[10] D. Patterson, G. Gibson and R. Katz, ‘‘A Case for Redundant Arrays of Inexpensive Disks (RAID)’’, *ACM SIGMOD 88*, Chicago, June 1988, 109-116.

[11] P. V. Rangan and H. M. Vin, ‘‘Designing File Systems For Digital Audio and Video’’, *Proc. of the 13th ACM Symp. on Operating System Prin.*, Pacific Grove, California, Oct. 1991, 81-94.

[12] J. M. Roth, G. S. Kendall and S. L. Decker, ‘‘A Network Sound System for UNIX’’, *Proceedings of the 1985 International Computer Music Conference*, Burnaby, B.C., Canada, Aug. 19-22, 1985, 61-67.

[13] D. Steinberg and T. Learmont, ‘‘The Multimedia File System’’, *Proc. 1989 International Computer Music Conference*, Columbus, Ohio, Nov. 2-3, 1989, 307-311.

[14] D. B. Terry and D. C. Swinehart, ‘‘Managing Stored Voice in the Etherphone System’’, *Trans. Computer Systems 6*, 1 (Feb. 1988), 3-27.

[15] C. Yu, W. Sun, D. Bitton, R. Bruno and J. Tullis, ‘‘Efficient Placement of Audio Data on Optical Disks for Real-Time Applications’’, *Comm. of the ACM 32*, 7 (1989), 862-871.

APPENDIX

Proof of Claim 2: We will show a client behavior that is consistent with the Read Session Axioms and that satisfies the above definition of reading a bounded-rate file in real time. Let the client behave as follows. At time i/R (for each $i \geq 0$) the client removes the next byte n from the FIFO if $T(n) < (i+1)/R$. It then waits until time $(i+1)/R$. Hence in each ‘‘time slot’’ of length $1/R$ the client either reads a byte or skips to the next time slot. The clock $C(t)$ advances during a read slot and pauses during a skip slot. The client ‘‘works ahead’’ by at most one byte, so (since the workahead parameter $\overline{Y}$ of the session is at least one) the Read Session Axioms are obeyed.

To verify the claim, we must first show that each byte n is read no later than $T(n) + E/R$. Suppose otherwise. Then some byte n is read at time j/R with

$$j/R > T(n) + E/R \tag{18}$$

If no skips occurred in slots $0 \cdots j$ then let $i = 0$; otherwise let i be such that $i-1$ is the last skip slot before j. Let m be the index of the byte read in slot i. Then

$$T(m) > i/R \tag{19}$$

since otherwise m would have been read in slot $i-1$. Now combining Eqs. 18 and 19 we have

$$T(n) - T(m) < (j - i - E)/R \tag{20}$$

Since the client reads in every slot from i to j we have $j - i = n - m$. Therefore

$$n - m > (T(n) - T(m))R + E \tag{21}$$

which contradicts the assumption that F is a bounded-rate file.

Finally we must show that at any time t, no more than E bytes i such that $T(i) > t - E/R$ have been read. Let i be such that $T(i) > t - E/R$. Then, given our specification of client behavior, byte i must have been read at time $t - E/R$ or later. The client reads at most E bytes in time E/R, so the claim holds. □