Real-Time Mutable Broadcast Disks*

Sanjoy Baruah
sanjoy@cs.uvm.edu
Department of CS & EE
University of Vermont

Azer Bestavros
best@cs.bu.edu
CS Department
Boston University

May 1997

Abstract

There is an increased interest in using broadcast disks to support mobile access to real-time databases. However, previous work has only considered the design of real-time immutable broadcast disks, the contents of which do not change over time. This paper considers the design of programs for real-time mutable broadcast disks — broadcast disks whose contents are occasionally updated. Recent scheduling-theoretic results relating to pinwheel scheduling and pfair scheduling are used to design algorithms for the efficient generation of real-time mutable broadcast disk programs.

Keywords: Broadcast disks; real-time constraints; disk updates; pfair scheduling; real-time database and information retrieval systems; mobile computing.

1 Introduction

Mobile computers are likely to play an important role at the extremities of future large-scale distributed real-time databases. One such example is the use of on-board automotive navigational systems that interact with the database of an Intelligent Vehicle Highway System (IVHS). IVHS systems allow for automated route guidance and automated rerouting around traffic incidents by allowing the mobile vehicle software to query and react to changes in IVHS databases [18, 17]. Other examples include wearable computers for soldiers in the battlefield and computerized cable boxes for future interactive TV networks and video-on-demand. Such systems are characterized by the significant discrepancy between the *downstream* communication capacity from servers (*e.g.* IVHS backbone) to clients (*e.g.* vehicles) and the *upstream* communication capacity from clients to servers. This discrepancy is the result

*This work has been partially supported by the NSF (grants CCR-9706685 and CCR-9596282).

of: (1) the huge disparity between the transmission capabilities of clients and servers (*e.g.*, broadcasting via satellite from IVHS backbone to vehicles as opposed to cellular modem communication from vehicles to IVHS backbone), and (2) the scale of information flow (*e.g.*, thousands of clients may be connecting to a single computer for service). Moreover, the limited power capacity of some mobile systems (*e.g.*, wearable computers) requires them to have no secondary I/O devices and to have only a small buffer space (relative to the size of the database) that acts as a cache for the information system to which the mobile system is attached.

Broadcast Disks: The concept of *Broadcast Disks* (Bdisks) was introduced by Zdonik *et al.* [27] as a mechanism that uses communication bandwidth to emulate a storage device (or a memory hierarchy in general) for mobile clients of a database system. The basic idea (illustrated in figure 1) is to exploit the abundant bandwidth capacity available from a server to its clients by *continuously and repeatedly* broadcasting data to clients, thus in effect making the broadcast channel act as *a set of disks* (hence the term "Broadcast Disks") from which clients could fetch data *"as it goes by."* Work on Bdisks is different from previous work in both wired and wireless networks [14, 16] in that several sources of data are multiplexed and broadcast to clients, thus creating a hierarchy of Bdisks with different sizes and speeds. On the server side, this hierarchy gives rise to memory management issues (*e.g.*, allocation of data to Bdisks based on priority/urgency). On the client side, this hierarchy gives rise to cache management and prefetching issues (e.g., cache replacement strategies to improve the hit ratio or reduce miss penalty). In [4], Acharya, Franklin and Zdonik discuss Bdisks organization issues, including client cache management [1], client-initiated prefetching to improve the communication latency for database access systems [3], and techniques for disseminating updates [2].

Previous work in Bdisks technology was driven by wireless applications and has concentrated on solving the problems associated with the limited number of uplink channels shared amongst a multitude of clients, or the problems associated with elective disconnection (as an extreme case of asymmetric communication), when a remote (*e.g.* mobile) client computer system must pre-load its cache before disconnecting. Problems that arise when timing *and* reliability constraints are imposed on the system were not considered.

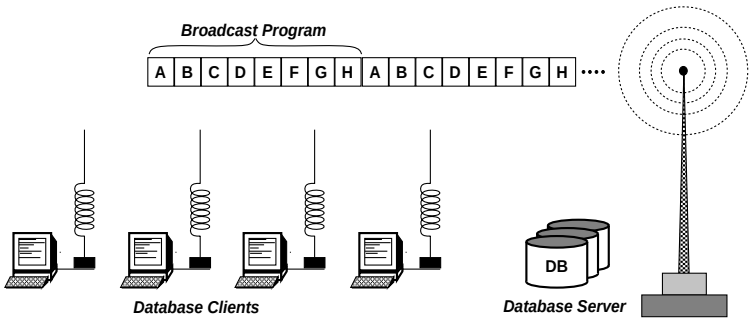

Figure 1: The Concept of Broadcast Disks

Real-time considerations: Previous work on Bdisks protocols has assumed that the rate at which a data item (say a page) is broadcast is dependent on the *demand* for that data item. Thus, *hot* data items would be placed on fast-spinning disks (*i.e.* broadcast at a higher rate), whereas *cold* data items would be placed on slow-spinning disks (*i.e.* broadcast at a lower rate). Such a strategy is optimal in the sense that it minimizes the average latency amongst all clients over all data items. In a real-time database environment, minimizing the average latency ceases to be the main performance criterion. Rather, guaranteeing (either deterministically or probabilistically) that timing constraints imposed on data retrieval will be met becomes the overriding concern.

There are many reasons for subjecting Bdisk data retrieval to timing constraints. Perhaps the most compelling is due to the absolute temporal consistency constraints [24] that may be imposed on data objects. For example, the data item in an Airborne Warning and Control System (AWACS) recording the position of an aircraft with a velocity of 900 km/hour may be subject to an absolute temporal consistency constraint of 400 msecs, in order to ensure a positional accuracy of 100 meters for client transactions (*e.g.* active transactions that are fired up to warn soldiers to take shelter). Notice that not all database object will have the same temporal consistency constraint. For example, the constraint would only be $6,000$ msecs for the data item recording the position of a tank with a velocity of 60 km/hour. Other reasons for imposing timing constraints on data retrieval from a Bdisk are due to the requirements of database protocols for admission control [12], concurrency control, transaction scheduling [23], recovery [15], and bounded imprecision [21, 25].

Bestavros [11] and Baruah & Bestavros [7] have defined a generalized model for real-time fault-tolerant Bdisks, that also incorporates consideration of the effect of transient failures

upon the real-time properties of Bdisks. They have shown that designing programs for Bdisks specified in this model is closely related to the scheduling of pinwheel task systems, and have exploited this relationship to design algorithms for the efficient generation of real-time fault-tolerant Bdisks programs.

A model for Broadcast Disks: We model a Bdisks system as being comprised of a set of data items (or files) that must be transmitted continuously and periodically to the client population. Each data item consists of a number of blocks. A block is the basic, indivisible unit of broadcast (*e.g.*, page). We assume that the retrieval of a data item by a client is subject to a time constraint imposed by the real-time process that needs that data item.

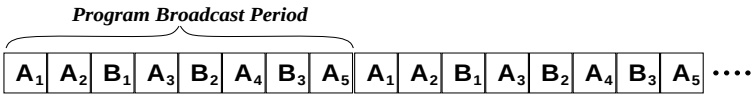

Figure 2: A flat broadcast program

Figure 2 illustrates a simple example of a flat broadcast program in which two files A and B are transmitted periodically by scanning through their respective blocks. In particular, file A consists of 5 blocks $A_1, \ldots, A_5$ and file B consists of 3 blocks $B_1, \ldots, B_3$.

Real-Time Mutable Broadcast Disks: As a general rule, the data to be broadcast on Bdisks is not completely static over time, but needs to be occasionally updated. Previous studies on designing program schedules for real-time Bdisks—namely the techniques of Bestavros in [11], Baruah and Bestavros in [7], and Xuang *et al* in [26]—have tended to ignore the issue of updates;[1] as a consequence, timeliness guarantees are compromised during an update. That is, if schemes based upon these previous studies are used for the design of broadcast programs, the real-time guarantees that are extended by these schemes hold only in the "steady state", when there are no updates—while a file is being updated, the latency guarantee with respect to that particular file is not honoured.

[1] The work of Acharya, Franklin, and Zdonick in [2] has considered the problem of disseminating updates in Bdisk programs, but has done so without taking the real-time constraints imposed on Bdisk data into consideration.

This research: The purpose of this research is to design algorithms for the efficient generation of real-time Bdisk programs, that continue to offer timing guarantees during file updates. We term a real-time Bdisk system that offer such a capability a *real-time mutable Bdisk system.*

2 Proportionate Progress

The task of communicating a data item on a Bdisk, hereinafter refered to as a Bdisk file, subject to a timing constraints requires that the various blocks of that file be broadcast periodically. Such a **periodic broadcast task** T_i is characterized by a *period* $p_i \in \mathbf{N}$ and a *resource usage requirement* $e_i \in \mathbf{N}$, with the interpretation that the task T_i expects to be allocated the communication channel for e_i units of time in every interval $\{t | k \cdot p_i \leq t < (k+1) \cdot p_i\}$, for each $k \in \mathbf{N}$. Given an instance $\Phi = \{T_1, T_2, \ldots, T_n\}$ of n such periodic tasks, the **periodic scheduling problem** [20] is concerned with attempting to schedule these n tasks on a single resource (*e.g.* processor or communication channel) so as to satisfy the constraints of each task. Task preemption is permitted, but only at integral boundaries as dictated by the integral boundary constraint—for each integer $t \geq 0$, the resource must be allocated to exactly one task (or remain unallocated) over the entire time interval $[t, t+1)$ (we refer to this time interval as *time slot* t.)

Liu and Layland have shown [20] that $\sum_{T_i \in \Phi}(e_i/p_i) \leq 1$ is a necessary and sufficient condition for a system Φ of periodic tasks to have a periodic schedule; furthermore, the *earliest deadline first* scheduling algorithm (EDF) [13] has been proven to be an optimal scheduling algorithm.

Temporal fairness: The issue of *fairness* in resource-allocation and scheduling has recently been attracting considerable attention [5, 6, 8]. Motivated no doubt in part by applications, such as multimedia, which are characterized by fairly *"regular"* resource requirements over extended intervals, attempts have been made to formalize and characterize notions of temporal fairness. The concepts of *proportionate progress* and *pfairness* were introduced in [8] (see also [9]) to quantitatively measure the fairness of a schedule. We briefly review these ideas below. We start with some conventions:

- We adopt the standard notation of having $[a, b)$ denote the contiguous natural numbers $a, a+1, \ldots, b-1$.
- The real interval between time t and time $t+1$ (including t, excluding $t+1$) will be referred to as slot t, $t \in \mathbf{N}$.
- We will consider an instance that involves one resource and a set $\Phi = \{T_1, T_2, \ldots, T_n\}$ of n tasks sharing that single resource.
- Each task T_i has two integer attributes—a *period* p_i and an *resource usage requirement* e_i. We define the *weight* w_i of task T_i to be the ratio e_i/p_i. Furthermore, we assume $0 < w_i \leq 1$.

A *schedule* S for instance Φ is a function from the natural numbers $\{0, 1, 2, \ldots\}$ to $\{0, 1, \ldots, n\}$, with the interpretation that $S(t) = i,\ i \in \{1, \ldots, n\}$, if the resource is allocated to task T_i for slot t, and $S(t) = 0$ if the resource is unallocated during time-slot t. Schedule S is a *periodic schedule* if and only if

$$\forall k, T_i : k \in \mathbf{N}, T_i \in \Phi : |\{t | t \in [0, p_i \cdot k) \text{ and } S(t) = x\}| = e_i \cdot k \ .$$

That is, each task T_i is allocated exactly $k \cdot e_i$ slots during its first k periods, for all k.

Let us define the *lag* of a task T_i at time t with respect to schedule S, denoted $\mathsf{lag}(S, T_i, t)$, as follows:

$$\mathsf{lag}(S, T_i, t) \stackrel{\text{def}}{=} w_i \cdot t - |\{t' | t' \in [0, t) \text{ and } S(t') = i\}| \ .$$

The quantity $w_i \cdot t$ represents the amount of time for which task T_i *should* have been allocated the resource over $[0, t)$, and $|\{t' | t' \in [0, t) \text{ and } S(t') = i\}|$ is equal to the number of slots for which task x was actually scheduled during this interval.

Informally, a schedule displays proportionate progress (equivalently, satisfies *pfairness*, or is *pfair*) if at all integer time instants t and for all tasks T_i, the lag of task T_i at time t—the difference between the amount of time for which T_i should have been allocated the resource, and the amount of time for which it was allocated the resource—is strictly less than 1 in absolute value. More formally, schedule S is *pfair* if and only if

$$\forall T_i, t : T_i \in \Phi, t \in \mathbf{N} : -1 < \mathsf{lag}(S, T_i, t) < 1.$$

That is, a schedule is pfair if and only if it is never the case that any task T_i is overallocated or underallocated by an entire slot.

Pfairness is an extremely stringent form of fairness—indeed, it has been shown [8] that no stronger fairness can be guaranteed to be achievable for periodic task systems in general. (Consider a system of n identical tasks, each with weight $1/n$. The task that is scheduled at slot 0 has a lag $(-1+1/n)$ at time 1, and the one scheduled at slot $n-1$ has a lag $(1-1/n)$ at time $(n-1)$. By making n large, these lags can be made arbitrarily close to -1 and $+1$, respectively.) It was proven in [8] that pfair scheduling is a stronger requirement than periodic scheduling, in that any pfair schedule is periodic. The converse, however, is not generally true.

The concept of pfairness was initially introduced in the context of constructing periodic schedules for a system of periodic tasks on several identical processors—the *multiprocessor periodic scheduling problem* [19]. The following theorem was proved in [8]:

Theorem 1 A system of periodic tasks can be scheduled in a pfair manner on m processors[2] provided the weights of all the tasks sum to at most m.

As a special case, we obtain the following corollary with respect to scheduling on a single resource:

Corollary 1.1 Every system of periodic tasks Φ for which $(\sum_{T_i \in \Phi} w_i \leq 1)$ holds has a pfair schedule on a single resource.

In addition, an on-line scheduling algorithm—*Algorithm PF*—was presented and proven correct. This algorithm has a non-trivial priority scheme that requires $O\left(\sum_{\text{all } T_i} \lceil \log(p_i+1) \rceil\right)$ time to determine the m highest-priority tasks in the worst case. However, for the single resource case, Algorithm PF reduces to a simple variant of EDF-scheduling, and can be implemented using the heap-of-heaps data structure [22] in $O(\log n)$ time per time slot, where n is the number of tasks.

3 Representing Bdisk Programs as Pfair Schedules

Let a real-time broadcast file F_i be represented by two integer parameters: $F_i = (m_i, d_i)$, with the interpretation that it consists of m_i blocks, and any client wishing to retrieve this file must be able to do so within d_i block-times ("slots") of wanting to do so. We start with some definitions:

[2] Or resources in general.

1. A *broadcast program* P for a system of n broadcast files $F_1, F_2, \ldots, F_n$ is a function from the non-negative integers to $\{0, 1, \ldots, n\}$, with the interpretation that $P(t) = i$, $1 \leq i \leq n$, iff a block of file F_i is transmitted during time-slot t, and $P(t) = 0$ iff nothing is transmitted during time-slot t.

2. $P.i$ is the sequence of non-negative integers t for which $P(t) = i$.

3. Broadcast program P satisfies *broadcast file condition* $\mathsf{bc}(i, m_i, d_i)$ iff $P.i$ contains at least m_i out of every d_i consecutive non-negative integers.

4. Broadcast program P satisfies *pfair task condition* $\mathsf{pfc}(i, w_i)$ iff $P.i$ contains at least $\lfloor w_i \cdot t \rfloor$ and at most $\lceil w_i \cdot t \rceil$ of the integers $0, 1, \ldots, t-1$, for all t. That is, the slots labelled $P.i$ would comprise a pfair allocation to a task T_i with weight w_i.

5. Broadcast program P satisfies *a conjunct of (pfair task or broadcast file) conditions* iff it satisfies each individual condition.

6. Let S_1 and S_2 be (broadcast/ pfair/ conjunct) conditions. We say that $S_1 \Rightarrow S_2$ iff any broadcast program satisfying S_1 also satisfies S_2. We say $S_1 \equiv S_2$ iff $S_1 \Rightarrow S_2$ and $S_2 \Rightarrow S_1$.

It has previously been shown in [7] that generating a broadcast schedule for n broadcast files is exactly equivalent to constructing a schedule for a system of n pinwheel tasks. From this equivalence, and the relationship between pinwheel scheduling and pfair scheduling recently identified by Baruah and Lin [10], we obtain the following lemma:

Lemma 1 If $m_i \geq 2$, then

$$\mathsf{bc}(i, m_i, d_i) \Leftarrow \mathsf{pfc}(i, w_i)$$

for all $w_i \geq m_i/(d_i - 1)$.

The following theorem is a direct consequence:

Theorem 2 The problem of constructing a broadcast schedule for $F_1, F_2, \ldots, F_n$ can be solved by obtaining a solution to the following pfair scheduling problem: Determine a broadacst program that satisfies

$$\bigwedge_{i=1}^{n} \left(\mathsf{pfc}(i, m_i/(b_i - 1)) \right), \tag{1}$$

provided $m_i \geq 2$ for all i, $1 \leq i \leq n$.

Proof Sketch: Let P be a broadcast program satisfying $\wedge_{i=1}^{n}(\mathsf{pfc}(i, m_i/(b_i - 1))$. By definition, P satisfies each of the individual pfair task conditions $\mathsf{pfc}(i, m_i/(b_i - 1))$ for each i. Hence by Lemma 1, P satisfies each $\mathsf{bc}(i, m_i, d_i)$, and is thus a broadcast program for broadcast files $F_1, F_2, \ldots, F_n$. ■

Example 1 Consider a system of two broadcast files $F_1 = (6, 11)$ and $F_2 = (3, 10)$, for which a broadcast schedule needs to be constructed. In keeping with Theorem 2, we attempt to determine a pfair schedule for the system of two periodic tasks T_1 and T_2 having weights $w_1 = 6/(11-1) = 0.6$ and $w_2 = 3/(10-1) = 0.\overline{3}$ respectively. The initial portion of the pfair schedule produced by Algorithm PF on this periodic task system is given below:

slot:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
task:	1	2	1	1	2	1	1	2	1	2	1	1	2	1	0
slot:	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
task:	1	2	1	1	2	1	1	2	1	2	1	1	2	1	0

The reader may validate that any interval of 11 contiguous slots contains at least 6 1's, and that any interval of 10 contiguous slots contains at least 3 1's.

4 Updating broadcast files

As a general rule, the data broadcast on Bdisks are not completely static over time; rather they must be occasionally updated. Ideally, we would like such updating at the broadcasting server to occur transparently to the client, who should suffer no degradation in performance (in the form of tardy file-availability) during an update. To illustrate the issues involved, let us consider a concrete example. Suppose that a file $F_i = (m_i, d_i)$ is being downloaded by a client; after the client has downloaded $m_i - 1$ blocks (in the worst case, this could take $d_i - 1$ time slots), however, the broadcast server begins broadcasting the *updated* version of the file. The previously-stored $m_i - 1$ blocks are now useless to the client, who has to download m_i blocks of the updated file over the next d_i time slots, for a total delay of $2d_i - 1$ time slots.

We wish to design Bdisk programs with the following performance characteristics: *The access time for a file F_i is d_i time slots even during an update to file F_i.* We permit that a request for F_i during an update interval may be honoured with either the original file or

the updated one: of course, consistency requirements rule out a mix of some blocks of one and the rest of the other. In this paper, we do not consider the possibility of broadcasting differential updates to enable clients that retrieved a stale file to "correct" their copies without having to wait unduly for that correction. Such a process is entirely possible in our scheme, but is orthogonal to the issue of switching to new "versions" of broadcast files without compromising the timeliness guarantees for retrieval of such files from real-time Bdisks.

We assume that requests to update files are made at the Bdisk server relatively infrequently. Namely, the frequency of updating a broadcast file is much lower than the frequency of broadcasting that file. When such an update request is made, it is desirable that it be serviced "as soon as possible;" however, there is no hard deadline associated with when this update is actually performed. We will refer to a broadcast program for broadcast files F_1, F_2, ..., F_n as a *mutable broadcast program*, if it permits such updates for each of the files $F_1, F_2, \ldots, F_n$

Our approach towards designing mutable broadcast programs is to reserve some broadcast bandwidth for an update server, and to use this update server to broadcast both the old and the updated versions of a file while it is being updated. That is, suppose that we decide to update file F_i at a particular time instant:

1. All the slots reserved in the broadcast program for both F_i and the update server are immediately pressed into service, in broadcasting the next $m_i - 1$ blocks of the old version of F_i. This consumes $m_i - 1$ slots—let us suppose that x slots reserved for F_i, and $m_i - 1 - x$ slots reserved for the update server, were used during this process.

2. After these $m_i - 1$ blocks have been transmitted, blocks of the updated file begin to be transmitted, and these are cyclically transmitted (until the next update). For this purpose, the next x slots reserved for the update server, as well as all the slots reserved for F_i, are used.

3. After x slots of the update server have been used to transmit the updated version of F_i (as described above), the update server ceases broadcasting blocks from F_i, and is available to update other files. Thus, a total of $(m_i - 1 - x) + x = m_i - 1$ blocks of the update server are used.

We will see below that *the update server has available to it at least m_i slots out of any contiguous interval of d_i slots.* Assuming for the moment that this is true, it is not difficult to see that this update procedure is correct. For, suppose that the server begins updating file F_i at time-instant t_s (i.e., at the start of slot t_s).

- A client that had downloaded any blocks of F_i prior to t_s can download the remaining blocks of the old version of F_i by its deadline—this follows from the observation that each of the next $m_i - 1$ blocks of the old version that are transmitted are transmitted no later than they would have been in the absence of the update.
- Any client that decides after instant t_s to download F_i will choose to retrieve the updated version[3]. Once again, this can be done within d_i slots, since

    1. Over the interval $[t_s, t_s + d)$, at least $2m_i$ slots are available for transmitting the old and the updated versions of F_i; hence, at least $2m - (m-1) \quad = \quad m+1$ slots are available for transmitting blocks of the updated version of F_i.
    2. Since $m_i - 1$ blocks of the update server are used during the update process, the update server is not used after instant $t_s + d_i$; after this instant, blocks of F_i are transmitted only in the slots reserved for it. Hence, the update process definitely terminates by time $t_s + d_i$.
    3. Each block of the updated version of F_i that is transmitted during the update process (i.e., during interval $[t_s, t_s + d_i)$) is transmitted no *earlier* as a result of the update than they would have been if they had been transmitted only in the slots reserved for F_i.

We now address the issue of bandwidth allocation for the update server, in order that it have available to it at least m_i slots out of any contiguous interval of d_i slots, for all i. Let $\rho_i \stackrel{\text{def}}{=} m_i/(d_i - 1)$, and $\rho_{\max} \stackrel{\text{def}}{=} \max_{i=1}^{n}\{\rho_i\}$. As a consequence of Lemma 1 and the fact that $\rho_{\max} \geq m_i/(b_i - 1)$ for all i, $1 \leq i \leq n$, we may conclude that a broadcast schedule satisfying $\mathsf{pfc}(i, \rho_{\max})$ will satisfy $\mathsf{bc}(i, m_i, d_i)$ for all i. Hence, reserving bandwidth for a periodic task T_{n+1} with weight $w_{n+1} = \rho_{\max}$ should provide sufficient bandwidth for the update server:

[3]We assume that each block of the old version of a file that is tranmsitted during an update has a filed identifying it as an old version of a file currently being updated — a client not already committed to retrieving this old version will ignore such a block.

Theorem 3 The problem of constructing a mutable broadcast schedule for $F_1, F_2, \ldots, F_n$ can be solved by obtaining a solution to the following pfair scheduling problem: Determine a broadacst program that satisfies

$$\mathsf{pfc}(n+1, \rho_{\max}) \wedge \left(\bigwedge_{i=1}^{n} (\mathsf{pfc}(i, \rho_i) \right), \tag{2}$$

provided $m_i \geq 2$ for all i, $1 \leq i \leq n$.

A sufficient condition for the existence of mutable broadcast programs: Given a system of broadcast files $F_1, F_2, \ldots, F_n$, $F_i = (m_i, d_i)$, we wish to determine whether we can design a mutable broadcast program for this file system. By Theorem 3, this problem can be solved by determining a pfair schedule on a single resource for a periodic task system of $n+1$ tasks, with total weight $\rho_{\max} + \sum_{i=1}^{n} \rho_i$. By Corollary 1.1, this is possible, provided that $(\rho_{\max} + \sum_{i=1}^{n} \rho_i) \leq 1$; i.e.,

$$\rho_{\max} \leq 1 - \sum_{i=1}^{n} \rho_i \tag{3}$$

Inequality 3 provides a quick sufficient test for determining whether we can construct a mutable broadcast program for a given system of broadcast files.

Example 2 Consider a system of three broadcast files $F_1 = (3, 12)$, $F_2 = (2, 16)$, and $F_3 = (3, 13)$, for which a broadcast schedule needs to be constructed. To determine whether a mutable broadcast program can in fact be constructed for this system, we need to determine whether Inequality 3 holds. For this system, $\rho_1 = 3/11$, $\rho_2 = 2/15$,and $\rho_3 = 3/12$; hence, $\rho_{\max} = 3/11$. Since $3/11 < 1 - (3/11 + 2/15 + 3/12)$, we conclude that Inequality 3 does hold for this system. Hence, to determine a broadcast program for this system of broadcast files, we can, by Theorem 3, obtain a pfair schedule for the system of four tasks T_1, T_2, T_3, and T_4, with weights $w_1 = 3/11$, $w_2 = 2/15$, $w_3 = 3/12$, and $w_4 = 3/11$, respectively. The initial portion of the pfair schedule produced by Algorithm PF on this periodic task system is given below:

slot:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
task:	1	4	3	1	4	3	2	4	1	3	2	4	1	3	4
slot:	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
task:	1	3	2	4	1	3	0	4	1	3	4	1	2	3	1

The reader may verify that the broadcast conditions for each of the three files is satisfied, in that any interval of d_i contiguous slots contains at least m_i slots labelled i, for $i = 1, 2,$ and 3. The slots allocated to task T_4 are the ones reserved for use by the update server—the reader may also verify that these are sufficient to permit the updating of files with no impact on performance. We illustrate the update process, if, for example, file F_3 were to be updated starting at instant 12 ("$\hat{3}$" denotes blocks of the old file F_3, while "3" denotes blocks of the updated F_3):

slot:	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
block:		$\hat{3}$	$\hat{3}$		3		3		3				3				3

The next $m_3 - 1 = 2$ slots allocated to either F_3 (T_3) or the update server (T_4) are used to transmit blocks of the old version of F_3. Thus, blocks of the old version of F_3 are transmitted during slots 13 and 14. The quantity x, representing the number of F_3's slots used in doing so, is equal to one. Hence, the next one slot of the update server's (at time 18) also goes to transmit blocks of file F_3, and the update process ends at the end of slot 18. At the end of slot 18, any client thatis in the process of retriving file F_3 has either zero, one or two blocks of (the updated version of) the file; however, if there had been no update, the same client would have had exactly one block less. Assuming that the client was going to meet its deadline in the absence of an update, therefore, it follows that the deadline will be met now as well.

5 Update latency

Since the update procedure described in Section 4 requires that the update server be available when an update is initiated, this implies that only one update can be processed at a time. If some other file is being updated when an update request for file F_i arrives, then the time at which the update of F_i is initiated may be delayed. We define the *update latency* of an update request as the length of the time interval (in number of slots) between the slot at which an update request is made, and the slot at which the update is initiated. (Note that, since any client that begins downloading the file once the update has been initiated will get the updated version, the latency interval is measured until the beginning of the update process, rather than until the end.)

We make a couple of assumptions regarding the arrival of update requests at the server: (1) these requests arrive at the server at the beginning of time-slots, and are queued at the server until one is selected to be serviced; (2) there is never more than one update request for the same file queued at the server at any given time — if a new update request for a file arrives before the update process has been initiated for a prior update request of the same file, then the prior request is discarded.

When there is more than one update request queued at the server at any given time, the server can choose from among several service policies in determining the order of service. We study a couple of these policies below. We will look only at *work-conserving* policies—if the update server is available and there is a waiting update request, then some update request is immediately serviced.

FCFS: In this update policy, update requests are serviced in the order in which they arrive (ties—simultaneous arrivals—are broken arbitrarily). Under this policy, we claim that the worst-case update latency occurs when an update of the file with the largest size has just been initiated, and update requests for all other files (including this one) immediately arrive simultaneously. In that case, the file that is last in the queue will have the largest update latency, being required to wait for every other file's update to complete (twice for the largest file) before its own update can be initiated.

Let us assume that file F_j is this last file in the queue. Let $F_{\max}$ denote the file with the largest size: $m_{\max} \stackrel{\text{def}}{=} \max_{i=1}^{n}\{m_i\}$. Since an update of file F_i consumes $m_i - 1$ blocks allocated to the update server, the total number of blocks of the update server consumed before F_j's update can commence is equal to

$$(m_{\max} - 1) + \left(\sum_{i=1}^{n}(m_i - 1)\right) - (m_j - 1)$$

$$= \quad (m_{\max} - m_j) + \sum_{i=1}^{n}(m_i - 1).$$

Recall that the bandwidth allocated the update server is w_{n+1}. It can be shown that $(m_{\max} - m_j) + \sum_{i=1}^{n}(m_i - 1)$ blocks are allocated to the update server over an interval of length $1 \ + \ \lceil((m_{\max} - m_j) + \sum_{i=1}^{n}(m_i - 1))/w_{n+1}\rceil$. From Corollary 1.1, it follows that the largest value for w_{n+1} is $(1 - \sum_{i=1}^{n}\rho_i)$ (recall that ρ_i was defined to be equal to $m_i/(d_i - 1)$). Hence,

the update latency for file F_j is no more than

$$1 + \left\lceil \frac{m_{\max} - m_j + \sum_{i=1}^{n}(m_i - 1)}{1 - \sum_{i=1}^{n} \rho_i} \right\rceil . \tag{4}$$

SJF: In this update policy, smaller-sized files are updated before larger ones (ties—equi-sized files—are broken arbitrarily). While this update policy will have smaller update latency than the FCFS update policy, the update-latency for every file (other than the unique smallest-sized one, if there is one) cannot be bounded.[4]

Between these two extremes of FCFS and SJF, hybrid variants can be defined which incorporate some form of aging to the pure SJF, such that update requests that have been waiting for a long time tend see their priority increase. Another alternative would be to enforce a minimum inter-update time for SJF to allow for a bounded latency of updates.

6 Summary

The importance of real-time Bdisk technology for information retrieval stems from two important trends that are only likely to continue in the future: (1) With the advent of mobile computers and cellular communication, it is expected that most clients in large-scale distributed environments will have limited storage capacities, a limited upstream bandwidth (if any) for transferring information to servers, and a large downstream broadcast bandwidth for receiving information from servers. (2) The increasing reliance on large-scale information systems and databases in supporting decision making processes—whether initiated by humans (*e.g.* stock-market trading) or by computers (*e.g.* collision avoidance systems aboard future vehicles on IVHS) subjects the information retrieval process to stringent timing constraints on data retrieval.

Previous work on real-time broadcast disks (Bdisks) has ignored the important issue of accomodating updates. As a consequence, if schemes based upon these previous studies are used for the design of broadcast programs, the real-time guarantees that are extended by these schemes hold only in the "steady state", when there are no updates—while a file is being updated, the latency guarantee with respect to that particular file is not honoured. In this paper we have extended our previous pinwheel-based programming of real-time Bdisks

[4]To understand why this is the case, it suffices to consider a situation in which update-requests for the smallest-sized file are repeatedly made, forcing all update-requests for larger files to wait for ever.

[7] to allow for the support of mutable broadcast programs. In particular, we have defined a formal model for the specification of the real-time requirements for mutable broadcast disk files. We have shown a close link between the design of broadcast programs for such disks and the previously studied problems of pinwheel scheduling, proportionate progress, and pfair scheduling [8, 9]. These results enable the design of efficient Bdisk programming techniques in the presence of updates in Bdisk data.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *Proceedings of ACM SIGMOD conference*, San Jose, CA, May 1995.

[2] S. Acharya, M. Franklin, and S. Zdonik. Disseminating updates on broadcast disks. In *Proceedings of VLDB'96: The 1996 International Conference on Very Large Databases*, India, September 1996.

[3] S. Acharya, M. Franklin, and S. Zdonik. Prefetching from a broadcast disk. In *Proceedings of ICDE'96: The 1996 International Conference on Data Engineering*, New Orleans, Louisiana, March 1996.

[4] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Dissemination-based data delivery using broadcast disks. *IEEE Personal Communications*, 2(6), December 1995.

[5] M. Ajtai, J. Aspnes, M. Naor, Y. Rabani, L. Schulman, and O. Waarts. Fairness in scheduling. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1995.

[6] A. Bar-Noy, A. Mayer, B. Schieber, and M. Sudan. Guaranteeing fair service to persistent dependent tasks. *SIAM journal on Computing*, 1996. to appear.

[7] S. Baruah and A. Bestavros. Pinwheel scheduling for fault-tolerant broadcast disks in real-time database systems. In *Proceedings of the IEEE International Conference on Data Engineering*, Birmingham, UK, April 1997. Extended version available as Tech Report TR–1996–023.

[8] S. Baruah, N. Cohen, G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996. Extended Abstract in the Proceedings of the 1993 ACM Annual Symposium on the Theory of Computing.

[9] Sanjoy Baruah, J. Gehrke, and G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the Ninth International Parallel Processing Symposium*, April 1995. Extended version available via anonymous ftp from `ftp.cs.utexas.edu`, as Tech Report TR–95–02.

[10] Sanjoy Baruah and Shun Shii Lin. Pfair scheduling of generalized pinwheel task systems. Submitted, 1996.

[11] Azer Bestavros. AIDA-based real-time fault-tolerant broadcast disks. In *Proceedings of the Real-Time Technology and Applications Symposium*, June 1996.

[12] Azer Bestavros and Sue Nagy. Value-cognizant admission control for rtdbs. In *Proceedings of RTSS'96: The 16th IEEE Real-Time System Symposium*, Washington, DC, December 1996.

[13] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[14] David Gifford. Ploychannel systems for mass digital communication. *Communications of the ACM*, 33, February 1990.

[15] Jing Huang and Le Gruenwald. An update-frequency-valid-interval partition checkpoint technique for real-time main memory databases. In *Proceedings of RTDB'96: The 1996 Workshop on Real-Time Databases*, pages 135–143, Newport Beach, California, March 1996.

[16] T. Imielinski and B. Badrinath. Mobile wireless computing: Challenges in data management. *Communications of the ACM*, 37, October 1994.

[17] IVHS America. IVHS architecture development program: Interim status report, April 1994.

[18] R.K. Jurgen. Smart cars and highways go global. *IEEE Spectrum*, pages 26–37, May 1991.

[19] C. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary 37-60*, II:28–37, 1969.

[20] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[21] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *Computer magazine*, pages 58–68, May 1991.

[22] A. Mok. Task management techniques for enforcing ED scheduling on a periodic task set. In *Proc. 5th IEEE Workshop on Real-Time Software and Operating Systems*, Washington D.C., May 1988.

[23] Özgür Ulusoy and Alejandro Buchmann. Exploiting main memory dbms features to improve real-time concurrency protocols. *ACM SIGMOD Record*, 25(1), March 1996.

[24] Krithi Ramamritham. Real-time databases. *International journal of Distributed and Parallel Databases*, 1(2), 1993.

[25] V. Fay Wolfe, L. Cingiser DiPippo, and J. K. Black. Supporting concurrency, timing constraints and imprecision in objects. Technical Report TR94-230, University of Rhode Island, Computer Science Department, December 1994.

[26] P. Xuan, S. Sen, O.J. Gonzalez-Gomez, J. Fernandez, and K. Ramamritham. Broadcast on demand: Efficient and timely dissemination of data. In *Proceedings of the 3rd Real-time Technology and Applications Symposium*, Montreal, Quebec, 1997.

[27] S. Zdonik, M. Franklin, R. Alonso, and S. Acharya. Are 'disks in the air' just pie in the sky? In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.