Distrib. Comput. (2004) 14: ♣–♣
Digital Object Identifier (DOI) 10.1007/s00446-003-0101-0

DISTRIBUTED COMPUTING

Optimal smoothing schedules for real-time streams*

Yishay Mansour[1], **Boaz Patt-Shamir**[2], **Ofer Lapid**[2]

[1] Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel
(e-mail: mansour@math.tau.ac.il)
[2] Deptartment of Electrical Engineering, Tel-Aviv University, Tel-Aviv 69978, Israel
(e-mail: boaz@eng.tau.ac.il; ofer@eng.tau.ac.il)

Received: November 21, 2001 / Accepted: November 6, 2003

Abstract. We consider the problem of smoothing real-time streams (such as video streams), where the goal is to reproduce a variable-bandwidth stream remotely, while minimizing bandwidth cost, space requirement, and playback delay. We focus on *lossy* schedules, where data may be dropped due to limited bandwidth or space. We present the following results. First, we determine the optimal tradeoff between buffer space, smoothing delay, and link bandwidth for lossy smoothing schedules. Specifically, this means that if two of these parameters are given, we can precisely calculate the value for the third which minimizes data loss while avoiding resource wastage. The tradeoff is accomplished by a simple generic algorithm, that allows one some freedom in choosing which data to discard. This algorithm is very easy to implement both at the server and at the client, and it enjoys the nice property that only the server decides which data to discard, and the client needs only to reconstruct the stream.

In a second set of results we study the case where different parts of the data have different importance, modeled by assigning a real "weight" to each packet in the stream. For this setting we use competitive analysis, i.e., we compare the weight delivered by on-line algorithms to the weight of an optimal off-line schedule using the same resources. We prove that a natural greedy algorithm is 4-competitive. We also prove a lower bound of 1.23 on the competitive ratio of *any* deterministic on-line algorithm. Finally, we give a few experimental results which seem to indicate that smoothing is very effective in practice, and that the greedy algorithm performs very well in the weighted case.

1 Introduction

The essence of real-time communication is to reproduce a given data stream in a remote location; one of the main difficulties in doing it is that often, the bit rate of a real-time stream varies with time, while communication bandwidth is usually allocated in advanced, and hence it is basically fixed. This fundamental conflict between variable bandwidth requirement and constant bandwidth supply has many possible solutions:

- Degradation of service by "truncating" the stream to the link rate [7], or alternatively, reserving bandwidth for the peak rate, resulting in link under-utilization [13].
- Statistical multiplexing [12], relying on an assumed statistical independence of the bit rates of different streams;
- More complex traffic descriptors such as VBR [19], allowing for a more refined specification of bursts;
- Renegotiation protocols [9] which facilitate dynamic bandwidth allocation; and
- *Smoothing,* which is the focus of this paper.

In smoothing, the basic idea is to trade bandwidth for space and latency. Specifically, the input stream, instead of being submitted directly to the communication link, is first stored in a *server's buffer* (see Fig. 1); the server submits data stored in its buffer to the link when it deems appropriate, subject to the link rate constraint. Data arriving at the other side of the link is first stored in a *client's buffer,* which delivers it to the playout device after a reconstruction action. The added flexibility provided by the buffers is used to create smoother traffic on the communication link, thus reducing the peak bandwidth requirement of the stream. Indeed, this technique is used on many levels, including, for example, MPEG encoders and decoders [2,3]. The drawbacks of this method are the additional memory space required, and the added latency. In many practical cases, however, one can significantly reduce the peak bandwidth using only a relatively modest amount of space without unbearable delay.

The basic problem in smoothing is determining what is the link rate, buffer sizes, the playout delay, and, of course, what is the right algorithm to use. Much is known about the off-line case, where the entire stream is given ahead of time, thus allowing for preprocessing. The off-line scenario is well justified in some cases, e.g., stored video. In many other settings, such as live broadcasts, one would like to smooth an unknown stream in an on-line fashion.

Our results. In this paper we consider a basic variant of the smoothing problem, where the link is lossless, its rate is constant, and consequently, data loss may occur. We assume that data is partitioned into units called "slices" that can be dropped

* Research supported in part by Israel Ministry of Science. An extended abstract of this paper appeared in *Proc. 19th ACM Symp. on Principles of Distributed Computing*, July 2000.

independently [6]. In some cases, called *fixed size slices*, all slices consist of the same number of bytes; otherwise, we say that the slices have *variable size*. Our first result is as follows. Denote the total smoothing delay at the server and the client by D, the buffer size (of the client and the server) by B, and the link rate by R. Suppose that two of the three parameters (space, delay and link rate) are given. We show that, for fixed size slices, the minimal number of slices is lost when the third parameter is set so as to satisfy the relation $B = R \cdot D$. (This result has a more refined form for variable size slices.) This result does not depend on any statistical assumptions: it holds deterministically, for all input streams. We explain how setting the free parameter such that $B \neq RD$ may adversely affect system performance in the sense that there exists a scenario in which there is either unnecessary loss of data or plain resource wastage.

The algorithm which achieves the minimal data loss is *generic*, in the sense that when some data must be dropped, it is not important which slices are discarded (provided they are available at the server). This intentional under-specification is very useful for practical encoding schemes, were the quality of the output does not degrade linearly with the quantity of lost data; in other words, the server is free to discard what seems to be the least damaging data. On the other hand, the client's algorithm does not involve such difficult decisions at all; the client's main burden is allocating the buffer space and reconstructing the stream.

We then consider the more general case, where different parts of the data have different importance. We model this case by assigning a positive real number called *weight* to each slice. The goal of a schedule in this generalized model is to maximize the *benefit*, defined to be the sum of the weights of slices delivered on time. For this setting, we use competitive analysis, i.e., we compare on-line algorithms to optimal off-line schedules. Competitive analysis is attractive because it avoids any statistical assumptions, and gives a performance bound that holds for any input sequence (see [4] for an introduction to competitive analysis). Specifically, we define the *competitive ratio* of an on-line algorithm to be the least upper bound, over all input sequences, of the benefit obtained by an optimal (off-line) schedule, divided by the benefit obtained by the on-line algorithm. For this setting, we have the following results. We present a natural greedy on-line algorithm for smoothing and prove an upper bound of 4 and a lower bound of almost 2 on its competitive ratio. In other words, we show that for any input stream, an optimal off-line schedule delivers at most 4 times the weight delivered by our algorithm, and that in some cases, the greedy algorithm delivers only a little more than half of the weight deliverable by an optimal off-line algorithm. (These results also have a more refined form for variable size slices.) We conclude our competitive analysis with a general lower bound of 1.23 on the competitive ratio of deterministic on-line algorithm.

To get some feeling for real-world data, we present in Sect. 5 a set of experimental results that seem to demonstrate the viability of smoothing: using MPEG clips found on the Internet, we show that modest buffer space can provide significant improvement in the number of dropped frames, and that the greedy algorithm usually has excellent performance in practice.

We close this paper in Sect. 6 with concluding remarks and a few open problems.

Related work on smoothing. A considerable body of research was devoted to minimizing the number and cost of changes for lossless smoothing. Rexford and Towsley [15] give a thorough study of lossless smoothing over an internetwork. In this setting, the output stream need not be identical to the input stream. Salehi *et al.* [16] present an off-line lossless smoothing algorithm which is optimal in terms of rate variability. Rexford *et al.* [14] extend the optimal off-line algorithm of [16] to the on-line case by applying the off-line algorithm piecewise, using a sliding window technique. This idea is refined in [5] by dynamically changing the sliding window size as a function of the current burstiness of the input stream. Zhao *et al.* [23] consider the effect of the initial delay in lossless schedules, and give an efficient algorithm that finds the optimal initial delay, after which there is no reduction in the peak bandwidth. The significance of the initial delay is also the motivation for the work of Sen *et al.* [18], where they investigate the use of a proxy server to cache a prefix of popular streams. This idea cannot be applied to on-line streams. Sen *et al.* [17] explore the tradeoff between minimum client startup delay, minimum client buffer requirement, transmission bandwidth, and channel holding time for CBR transmission of VBR video. Jiang and Kleinrock [11] give an optimal off-line algorithm for the problem of minimizing the total cost of changes of a lossless smoothing schedule. Their only assumption is that the cost of a change is local, in the sense that the overall cost is the sum of the individual change costs.

Another research direction is to study the inherent properties of VBR traffic, and video in particular. For example, Wrege *et al.* [20] present a deterministic model for VBR traffic, and study the earliest-deadline-first scheduling policy in this context. Feng and Rexford [8] present an experimental study of various smoothing algorithms. Duffield *et al.* [7] propose an algorithm for controlling the MPEG encoding so as to get smooth streams. Ni *et al.* [10] present a formula similar to ours for the connection between delay, buffer size, and link rate. However, their model is different: all the input stream is available in advance, so different bytes may have different smoothing delays. In their model, they claim that the delay-bandwidth product is a lower bound on the buffer size. Zhang *et al.* [22] study the case where the buffer size, playout delay and the link rate are given, and the question is how to minimize the number of *frames* discarded, under the assumption that only whole frames can be discarded. They give an optimal algorithm, and a few heuristics which appear to be quite efficient for motion-JPEG streams.

Paper organization. The rest of this paper is organized as follows. In Sect. 2 we describe the underlying model in detail. In Sect. 3 we present the generic algorithm and prove the basic connection between delay, buffer space, and link rate. In Sect. 4 we consider the model where each byte may have a different weight. Finally, in Sect. 5, we present some performance results for weighted and unweighted lossy smoothing. We summarize in Sect. 6.

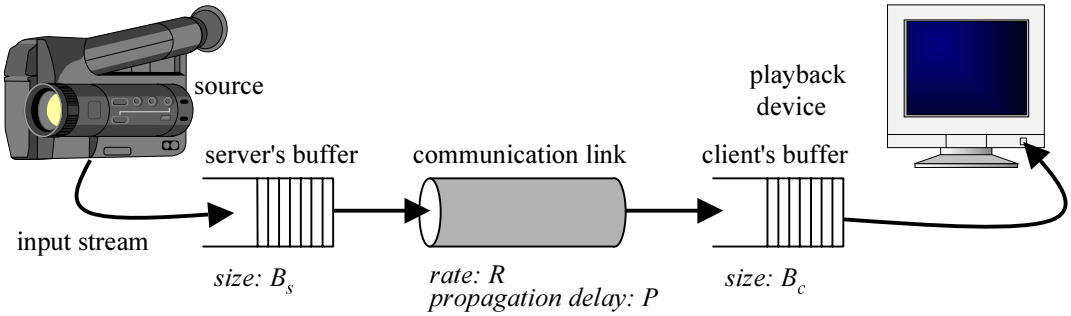

Fig. 1. A schematic representation of a smoothing system. The link is lossless and FIFO

2 Problem statement

2.1 Informal description

Intuitively, the system we consider consists of a source, a server buffer, a communication link, a client buffer, and a sink (see Fig. 1). During each time step (we consider a slotted time model), the following events happen.

- A *source* generates some data called a *frame*, which is inserted to a *server's buffer*. Each frame is a collection of *slices*, and each slice is a sequence of bytes. Slices are the basic units of data that can be dropped individually without affecting other slices [6], and bytes are the basic data units that can be sent over the link individually. (Throughout this paper, we use the term "byte" in an abstract sense: all that our model requires is that all "bytes" have equal size, and that they can be sent individually over the link.)
- The *server* submits some data currently stored in its buffer to a *link*.
- The *link* delivers the data it holds to the *client's buffer*. The link is assumed to be lossless, first in first out (FIFO), and with a constant delay per byte.
- The *client* plays out some of the complete slices stored in its buffer by delivering them to a *playout device*.
- In addition, any number of the slices currently stored at the server's buffer or the client's buffer may be *dropped* at the discretion of the algorithm.

Slices may be dropped due to the following reasons.

Server overflow. The server's buffer has limited space, and the link has a limited rate which bounds the rate at which the server's buffer can be drained. If the size of an incoming frame is larger than the current free space in the buffer, some of the slices (possibly old) must be discarded. A slice cannot be dropped after it starts being transmitted (i.e., no pre-emption).

Client overflow. Usually, there is a limit on the rate in which the client's buffer is drained (this is always the case in real-time systems – see below). If the number of bytes arriving at the client is larger than its current free space, then some slices must be dropped.

Early drop. The algorithm may drop slices at any time, even when no overflow occurs, possibly to avoid drops later.

In our model, the buffers are *random access* (a.k.a. *push-out*) buffers, i.e., any part of the data stored in buffer can be removed to free its space.

The goal of a real-time smoothing schedule is to reconstruct the input stream generated at the source as closely as possible at the playout device. To this end, we assume that there is some convention by which we can recognize, for each piece of data, what is its arrival time (e.g., by a frame number carried in the header).

The performance measures of interest for a smoothing system are the following.

- Playout Latency: How long does it take for a byte since it is generated at the source until it is played out.
- Buffer Requirement: How much buffer space do the server and client need.
- Link Rate: What is the maximal link rate used by the smoothing schedule.
- Fidelity: How close is the reconstructed stream to the original input stream.

Note that while playout latency, buffer space and link rate are well defined and easy to measure, fidelity is harder to quantify. In particular, if the schedule is *lossy*, (i.e., some of the data is dropped), the perceived fidelity seems subjective [22], and depends on the actual stream and encoding scheme employed. In full generality, one may have a cost function assigning a "fidelity index" for the pair of full input and output streams. A simpler alternative is to have a *local* value function assigning a (potentially different) value to each slice, and the overall fidelity index is the sum of the value of played-out data. A special case of a local value function is assigning to each slice s its size $|s|$, thus simply measuring how many bytes are played out by the system.

2.2 Formal model and notation

The following definition formalizes the notion of an input stream. Intuitively, it is defined by specifying for each slice its exact time of arrival in the system, and its size in bytes. For example, a frame of a video stream is just the set of all slices whose arrival time is the time the frame is generated. Using slices as a basic entity will allow us to drop data at a finer level than frames. The lower-level notion of "bytes" allows for segmentation and re-assembly at the link level.

Definition 2.1 *An* input stream *is a set $\mathcal{B}$ of* slices*. Each slice s is a set of $|s| \in \mathbb{N}$* bytes*, and has its* arrival time $\mathsf{AT}(s) \in \mathbb{N}$*. The* size of a set B *of slices is $|B| \stackrel{\text{def}}{=} \sum_{s \in B} |s|$. The size of a set of bytes is the number of bytes it contains.*

(We use $\mathbb{N}$ to denote the set of natural numbers including 0.) In many cases, we abuse notation slightly and when talk about a "set of slices" we actually refer to the set of bytes contained in these slices. As we shall see later, a byte inherits most of its attributes from the slice that contains it. This is why the size of a set of slices is defined to be the number of bytes contained in its slices.

Next, we formally define the notion of a schedule. In the definition below, we use the convention that the time of an event is infinity if and only if the event never occurs.

Definition 2.2 *A* smoothing schedule *for a given input stream is defined by the following additional functions.*

- send time *function, denoted* ST*, mapping bytes to* $\mathbb{N} \cup \{\infty\}$.
- receive time *function, denoted* RT*, mapping bytes to* $\mathbb{N} \cup \{\infty\}$.
- playback time *function, denoted* PT*, mapping bytes or slices to* $\mathbb{N} \cup \{\infty\}$.
- drop time *function, denoted* DT*, mapping bytes or slices to* $\mathbb{N} \cup \{\infty\}$.

All bytes in a slice arrive at the same time, played back at the same time, and discarded at the same time (possibly infinite). However, bytes of the same slice may have different send times and different receive times. For example, all the bytes in a slice s are generated by the source at time $\mathsf{AT}(s)$. A byte b in slice s may then wait in the server's buffer until time $\mathsf{ST}(b)$, when it enters the link. At time $\mathsf{RT}(b)$ byte b is received at the client's buffer, where it waits until it is played out at time $\mathsf{PT}(b) = \mathsf{PT}(s)$. For this slice we have infinite drop time, i.e., $\mathsf{DT}(s) = \infty$. Another slice s' may be generated at the source at time $\mathsf{AT}(s')$, and dropped from the server's buffer at time $\mathsf{DT}(s')$; for this slice we have $\mathsf{ST}(s') = \mathsf{RT}(s') = \mathsf{PT}(s') = \infty$. Naturally, the functions defined above must obey certain restrictions if they describe a schedule as we intend, such as "a byte is not sent before it arrives" etc. We skip the formal details here.

The following additional derived notations are intuitive and useful.

Definition 2.3 *Let a smoothing schedule be given.*

- $\mathsf{A}(t)$ *is the set of slices (alternatively, bytes) arriving at time* t*, also called the* frame generated at time t.
- $\mathsf{S}(t)$ *is the set of bytes transmitted at time* t.
- $\mathsf{R}(t)$ *is the set of bytes delivered at time* t.
- $\mathsf{P}(t)$ *is the set of slices (alternatively, bytes) played out at time* t*, also called the* frame played at time t.
- $\mathsf{D}(t)$ *is the set of slices dropped at time* t.
- $\mathsf{B}_s(t)$ *is the set of bytes stored at the server at time* t.
- $\mathsf{B}_c(t)$ *is the set of bytes stored at the client at time* t.
- *The* sojourn time *for a byte (alternatively, a slice)* b *is* $\mathsf{PT}(b) - \mathsf{AT}(b)$.
- *The* link delay *for a byte* b *with finite sojourn time is* $\mathsf{RT}(b) - \mathsf{ST}(b)$.
- *The* smoothing delay *for a byte* b *with finite sojourn time is its sojourn time minus its link delay, i.e,* $(\mathsf{PT}(b) - \mathsf{AT}(b)) - (\mathsf{RT}(b) - \mathsf{ST}(b))$.

The first five functions are extended to hold for time intervals. For example, if I *is a time interval, then* $\mathsf{A}(I)$ *is the set of all bytes that arrive in* I*, i.e.,* $\mathsf{A}(I) = \bigcup_{t \in I} \mathsf{A}(t)$*, etc.*

Let us be more specific regarding the order of events at the server. The convention we use is as follows. Just before step t starts, the server stores $\mathsf{B}_s(t-1)$ and the client stores $\mathsf{B}_c(t-1)$. At step t, first the set $\mathsf{A}(t)$ arrives at the server. The server then may drop $\mathsf{D}(t) \subseteq \mathsf{A}(t) \cup \mathsf{B}_s(t-1)$, and then it sends $\mathsf{S}(t) \subseteq \mathsf{A}(t) \cup \mathsf{B}_s(t-1) \setminus \mathsf{D}(t)$. The remaining set at the server is $\mathsf{B}_s(t) = (\mathsf{A}(t) \cup \mathsf{B}_s(t-1)) \setminus (\mathsf{D}(t) \cup \mathsf{S}(t))$.

Finally, we define the performance metrics for a given schedule. Note that our measures are defined given the full schedule.

Definition 2.4 *Let a smoothing schedule be given.*

- *The* server buffer requirement *is the least upper bound on* $|\mathsf{B}_s(t)|$ *over all* t.
- *Similarly, the* client buffer requirement *is the least upper bound on* $|\mathsf{B}_c(t)|$ *over all* t.
- *The* link rate requirement *is the least upper bound on* $|\mathsf{R}(t)|$ *over all* t.
- *The* throughput *of a schedule is the total number of bytes played out by the schedule.*

We sometimes refer to buffer requirement as *space requirement*.

Thus far, we have defined a model for transmitting a general input stream with buffers. Our last step is to specialize the model to the case of *real-time* streams.

Definition 2.5 *A smoothing schedule is said to be a* real-time smoothing schedule *if the sojourn time for all slices with finite sojourn time is the same.*

For the most part of this paper, we abstract the communication network as a single lossless link with 0-jitter, i.e., the link delay of all bytes is some fixed parameter P. This simplification (that can be justified by jitter control algorithms, e.g., [21]) allows us to concentrate on smoothing without taking into account the (important) effect of delay jitter. More specifically, with this abstraction, we assume that *smoothing delay* occurs only in the server and the client. Note that for the case of real time schedules with 0 delay jitter, all bytes in non-dropped slices must have the same smoothing delay: bytes belonging to the same slice arrive and are played back at the same time, and they spend the same amount of time in transit over the link. We denote this common smoothing delay by D.

We close this section with a definition of a local *weight* (sometimes called *value*) function, which is a simplified representation of the fact that not all slices in a stream have equal importance. The function is called "local" since the value assigned to a slice does not depend on other slices.

Definition 2.6 *A* local weight function w *for a given input stream* $\mathcal{B}$ *is a function which assigns a real number* $w(s)$ *to each slice* $s \in \mathcal{B}$*. The* benefit *of a real-time schedule* S *for* $\mathcal{B}$ *is the sum of weights of all slices with finite sojourn time.*

Note that the benefit induced by the local weight function which assigns $|s|$ to each slice s is simply the throughput of the schedule.

3 The generic algorithm

In this section we consider schedules whose goal is to maximize the *number* of slices played out by the client. Our results

hold for the general case of slices with different sizes. We first assume that all slices have size 1. In Sect. 3.1 we present the algorithm, and in Sect. 3.2 we show that it is optimal for the special case of unit-size slices. We then extend the results to the general case, where slices may have different sizes. For this case, we prove that the algorithm is guaranteed only to be close to optimal (the ratio between the algorithm and the optimal value depends on the size of the largest slice). The latter proof contains a result relating the throughput of *buffers* of different sizes (and unit size slices), which may be of independent interest.

3.1 The algorithm

Recall that in the case of real-time schedules with 0 delay jitter, all slices have exactly the same smoothing delay D. The algorithm is defined by means of the following quantities: maximum line rate R, smoothing delay D, and buffer space $B = \min(\mathsf{B}_s, \mathsf{B}_c)$. Our main result in this section (Theorem 3.5) is that when given any two of these quantities, the optimal choice for the third is determined by the identity

$$B = D \cdot R \tag{1}$$

The buffer space needed at the client and the server is equal to B: making only one of the buffers bigger does not help, as we shall show later.

We remark that this result is the first proof that relates the three quantities in the case of lossy on-line scheduling algorithms.

3.1.1 Server's algorithm

The server's job is extremely simple: whenever the server's buffer is non-empty, its contents is transmitted, in FIFO order, to the client at the maximal possible rate. Whenever a new frame is input into the server's buffer, the size of the buffer is checked to ensure that the total number of slices in the server's buffer does not exceed B. If the server's occupancy becomes $B + Z$ for some $Z > 0$, then *an arbitrary set of Z slices is discarded from the buffer*. We leave the way information is discarded unspecified at this point. Recall, however, that a slice is never pre-empted after it commenced transmission (this is applicable to the case of variable slice sizes).

Formally, to describe our schedule, recall that $\mathsf{S}(t)$, $\mathsf{B}_s(t)$ and $\mathsf{D}(t)$ denote the set of bytes sent, stored or dropped, respectively, at time t. The algorithm specification for any step t is as follows:

$$|\mathsf{S}(t)| = \min\Big(R\,,\, |\mathsf{B}_s(t-1)| + |\mathsf{A}(t)|\Big) \tag{2}$$

$$|\mathsf{D}(t)| = \max\Big(0\,,|\mathsf{B}_s(t-1)|+|\mathsf{A}(t)|-|\mathsf{S}(t)|-B\Big) \tag{3}$$

The actual identity of the slices dropped is unrestricted, provided they are available at the server at time t:

$$\mathsf{D}(t) \subseteq \mathsf{B}_s(t-1) \cup \mathsf{A}(t) - \mathsf{S}(t)\ .$$

For the set of transmitted bytes, we require that these are the slices with the smallest arrival times among all slices which were not dropped (i.e., the server buffer maintains FIFO ordering).

3.1.2 Client's algorithm

The client's algorithm is even simpler: when the first slice s_0 arrives at the client's buffer, a timer is set to D time units. When the timer expires, all available slices that arrived in the system at time $\mathsf{AT}(s_0)$ (i.e, all the slices of the first frame) currently stored in the client's buffer are played out; thereafter, at each step t, frame t—that is, the slices of frame t—are displayed. (We show later that only whole slices are played out.) Formally, we have

$$\mathsf{P}(t) = \{s\ :\ \mathsf{AT}(s) = t-P-D,\ \mathsf{RT}(s) \le t\}\ .$$

3.2 Analysis for fixed size slices

We now analyze the generic algorithm for the case that $|s| = 1$ for all slices s. Specifically, we prove that the generic algorithm above loses the least number of slices (alternatively, has the maximal throughput) among all algorithms with buffer space B and link rate R.

For the remainder of this section, fix the input stream. To show optimality of the algorithm, we first state the simple fact that the server transmits as many slices as possible, subject to the server buffer and link rate constraints. This fact follows from the greedy nature of the server's algorithm.

Lemma 3.1 *Let $\mathsf{S}(t)$ denote the set of bytes transmitted by the generic algorithm at time t for a given input stream, and let $\hat{\mathsf{S}}(t)$ denote the set of bytes transmitted by any schedule for that stream using server buffer size B and link rate R. Then for all t, $\sum_{i=0}^{t} |\mathsf{S}(i)| \ge \sum_{i=0}^{t} |\hat{\mathsf{S}}(i)|$.*

Since the server is pushing as many bytes as possible to the link, it is sufficient to show that all bytes which arrive at the client are played out. We prove this fact by showing that there is no client overflow, and that no byte arrives at the client too late.

We first bound the space requirement of the server, and the time any byte spends in the server's buffer.

Lemma 3.2 *The server's buffer requirement under the schedules produced by the generic algorithm is B. Moreover, no byte is submitted to the link more than B/R time units after its arrival.*

Proof. The server's buffer requirement is obvious from the algorithm: at any time t, the number of bytes dropped, according to Eq. (3) is exactly the minimal number which ensures that $|\mathsf{B}_s(t)| \le B$. For the second part of the lemma, note that by the FIFO order of transmission, and since by Eq. (2) the server transmits at maximum rate whenever it is not empty, a byte which arrives at time t is submitted to the link by time $t + |\mathsf{B}_s(t)|/R$, or else it was dropped by that time. □

A direct implication of Lemma 3.2 is that no byte arrives at the client too late (recall that P is the link's constant delay).

Lemma 3.3 *Let t be any time step. For all bytes $b \in \mathsf{R}(t)$, we have that $\mathsf{AT}(b) \ge t - (P+B/R)$. Conversely, for any byte b: if $\mathsf{AT}(b) = t$ and b is not dropped by the server, then $t+P \le \mathsf{RT}(b) \le t+P+B/R$.*

Proof. Follows immediately from Lemma 3.2, and the assumption that there is a constant delay of P time units over the link, i.e., $\mathsf{R}(t) = \mathsf{S}(t-P)$. □

Lemma 3.3 shows that no slice has to be discarded by the client due to missing its deadline, i.e., there is no *underflow*. Next, we prove that there is no overflow at the client buffer.

Lemma 3.4 *The buffer space requirement of the client under the generic algorithm is B.*

Proof. Consider any time step t. Note that all bytes stored in the client's buffer at time t were input to the system after time $t-P-B/R$: bytes with smaller arrival time have already been played out by the client's algorithm, or have been discarded by the server by Lemma 3.3. Also, since we assume that all bytes have a fixed link delay P, we have that all bytes stored at the client's buffer at time t were input no later than time $t-P$. Formally, for all $b \in \mathsf{B}_c(t)$, we have that $t-P-\frac{B}{R} \leq \mathsf{AT}(b) \leq t-P$. This, using Lemma 3.3, means that all bytes stored at the client at time t were delivered by the link in the time interval $[t-B/R\,,\,t]$. Therefore, since the link delivers at most R bytes at each time step, we conclude that $|\mathsf{B}_c(t)| \leq B$. □

Thus we have the following result, which is a direct consequence of the lemmas above.

Theorem 3.5 *Let $\mathsf{D}(t)$ denote the set of slices dropped by the generic algorithm at time t for a given input stream, and let $\hat{\mathsf{D}}(t)$ denote the set of slices dropped by any real-time schedule for that stream using server buffer size B and link rate R. Then for all t, $\sum_{i=0}^{t} |\mathsf{D}(i)| \leq \sum_{i=0}^{t} |\hat{\mathsf{D}}(i)|$.*

3.3 Practical considerations

Let us make a few remarks about our smoothing protocol from the practical viewpoint. First, note that the algorithm works without explicit clock synchronization: the client just sets the timer to D when the first slice arrives; when this timer goes off, the client starts playing out one frame at a step (more precisely, the slices that were not dropped from that frame). Thereafter, synchronization is implied by the fact that the source and the playout device run at the exact same rate.

Secondly, it is straightforward to use the result in a simple setup protocol: the client and the server advertise their buffer size in the connection setup message. A client may also specify the desired latency, from which the required bandwidth can be computed.

It is interesting to note what happens if $B \neq RD$. The following observations provide some insight:

1. Suppose $B < RD$. Let us consider separately what happens if either B, R or D can be changed by the algorithm.
   - If B and R are fixed and D is under the algorithm's control, we may decrease the playout delay to any value larger than B/R, and the number of dropped slices will not increase if we use the generic algorithm, for any input sequence. To see that, note that each byte spends at least $D - B/R > 0$ time units at the client's buffer: we may reduce this unnecessary delay to any non-negative value without incurring additional overflows. Moreover, if the client's buffer size was not large enough, then decreasing the delay may actually *increase* the throughput, by eliminating some of the client overflows.
   - If R and D are fixed and B can be changed, then in the case of server overflows, increasing B to any value not larger than DR will increase the total throughput without changing the delay and rate.
   - If B and D is fixed and R can be changed, the situation is different: on some inputs, decreasing the rate to B/D will result in less throughput. This happens, for example, when the input is perfectly smooth with rate $R > B/D$.
2. Suppose $B > DR$.
   - If B can be changed, we may decrease the size of the server buffer and client buffer to DR without increasing the number of dropped slices in any input sequence, by using the generic algorithm.
   - Similarly, if we increase either B or D we may increase throughput, so long as we maintain $DR \leq B$.

3.4 Analysis for variable size slices

We now extend the results above to the case where we have that the size of each slice is in the range $[1, L_{\max}]$ for some parameter $L_{\max}$. We show that in this case, the generic algorithm may not be optimal, but its throughput is very close to optimal: the degradation is no more than a factor of $(B-L_{\max}+1)/B$. The extension is done in two steps. First, we prove a general relation between the total throughput of two buffers of different sizes for unit-size slices (when the algorithm and the arrival sequence are fixed, it makes sense to discuss the throughput corresponding to a given buffer size). Then we show a simple reduction from arbitrary slice sizes to different server buffer sizes.

3.4.1 The throughput of different server buffer sizes

Let us ignore the client's buffer for a moment, and consider only the throughput of the server's buffer, i.e., the total number of slices submitted to the link. We have the following general result, bounding the throughput of a server buffer with respect to another server with a larger buffer, assuming that all slices have size 1.

Lemma 3.6 *Fix an input stream where all slices have size 1, and consider two schedules: $\mathcal{S}_1$ is a schedule produced by the generic algorithm using buffer of size B_1, and $\mathcal{S}_2$ is a schedule produced by the generic algorithm using buffer of size $B_2 \geq B_1$. Then the throughput of $\mathcal{S}_1$ is at least B_1/B_2 times the throughput of $\mathcal{S}_2$. Formally, let $T \geq 0$ be the last step where a slice is sent by either $\mathcal{S}_1$ or $\mathcal{S}_2$. Then*

$$\sum_{t=0}^{T} |\mathsf{S}_1(t)| \geq \frac{B_1}{B_2} \cdot \sum_{t=0}^{T} |\mathsf{S}_2(t)| \,.$$

Proof. Let $\mathsf{B}_1(t)$ denote the server's buffer contents at time t under $\mathcal{S}_1$, and similarly, let $\mathsf{B}_2(t)$ denote the server's buffer

contents at time t under $\mathcal{S}_2$. To prove the lemma, we consider some of the "busy periods" of $\mathcal{S}_1$. Formally, a *busy period* is an interval $I = [\mathrm{start}(I), \mathrm{end}(I)]$ such that $\mathsf{B}_1(\mathrm{start}(I)-1) = \mathsf{B}_1(\mathrm{end}(I)) = \emptyset$, and $\mathsf{B}_1(t) \neq \emptyset$ for all $t \in I \setminus \{\mathrm{end}(I)\}$. Let

$$\mathcal{I} = I : I \text{ is a busy period such that } |\mathsf{B}_1(t)| = B_1 \text{ for some } t \in I ,$$

namely $\mathcal{I}$ is the set of all busy periods in which the buffer of $\mathcal{S}_1$ is full at some point. Fix T from the statement of the lemma, and let $W = [0,T] - \bigcup_{I\in\mathcal{I}} I$, i.e., W is the set of all time steps not included in $\mathcal{I}$. We now claim the following.

1. $\mathcal{S}_1$ does not drop slices at time steps included in W.
2. For all $I \in \mathcal{I}$, we have that $|\mathsf{S}(I)| \geq B_1$.
3. For all $I \in \mathcal{I}$, we have that $\mathsf{B}_2(\mathrm{end}(I)) \leq B_2 - B_1$.

Claim 1 follows from the fact that when an overflow occurs in the generic algorithm, then the buffer is full. Claim 2 follows from the fact that $|\mathsf{B}_1(t)| = B_1$ for some $t \in I$. Claim 3 follows from showing, by induction on time, that $|\mathsf{B}_1(t)| \geq |\mathsf{B}_2(t)| - (B_2 - B_1)$ for all t: this is true because $\mathcal{S}_1$ discards slices only when its buffer occupancy is larger than B_1, and because the space requirement of $\mathcal{S}_2$ is never more than B_2.

We can now prove the lemma (see justifications below).

$$\begin{aligned}
\frac{\sum_{t=0}^{T}|\mathsf{S}_2(t)|}{\sum_{t=0}^{T}|\mathsf{S}_1(t)|} &= \frac{|\mathsf{S}_2(W)| + \sum_{I\in\mathcal{I}}|\mathsf{S}_2(I)|}{|\mathsf{S}_1(W)| + \sum_{I\in\mathcal{I}}|\mathsf{S}_1(I)|} && (4)\\
&\leq \frac{|\mathsf{A}(W)| + \sum_{I\in\mathcal{I}}\Big(|\mathsf{S}_2(I)| + |\mathsf{B}_2(\mathrm{end}(I))|\Big)}{|\mathsf{S}_1(W)| + \sum_{I\in\mathcal{I}}|\mathsf{S}_1(I)|}\\
&\leq \frac{\sum_{I\in\mathcal{I}}\Big(|\mathsf{S}_2(I)| + |\mathsf{B}_2(\mathrm{end}(I))|\Big)}{\sum_{I\in\mathcal{I}}|\mathsf{S}_1(I)|} && (5)\\
&\leq \frac{\sum_{I\in\mathcal{I}}|\mathsf{S}_2(I)|}{\sum_{I\in\mathcal{I}}|\mathsf{S}_1(I)|} + \frac{|\mathcal{I}|(B_2-B_1)}{|\mathcal{I}|(B_1)} && (6)\\
&\leq 1 + \frac{B_2 - B_1}{B_1} && (7)\\
&= \frac{B_2}{B_1} .
\end{aligned}$$

To see that Inequality (5) is true, note that W is a set of disjoint intervals whose start times are the end times of intervals in $\mathcal{I}$; the inequality follows because the set of slices transmitted by $\mathcal{S}_2$ in W is at most all slices that arrive in W plus the slices that reside in the buffer of $\mathcal{S}_2$ at the start of the intervals. Inequality (5) is justified since by Claim 1 and the definition of $\mathcal{I}$, we have that $|\mathsf{S}_1(W)| = |\mathsf{A}(W)|$. Inequality (6) follows from Claims 2 and 3. Inequality (7) follows from the fact that in a busy period, $\mathcal{S}_1$ transmits at the maximal possible speed, and thus for each $I \in \mathcal{I}$, we have that $\sum_{t\in I}|\mathsf{S}_2(t)| \leq \sum_{t\in I}|\mathsf{S}_1(t)|$. □

We remark that Lemma 3.6 is the best possible, in the sense that its inequality is met with equality on some cases. To see that, consider the input stream that consists of a batch of B_2 slices arriving at the same step, followed by B_2-1 steps with no arrivals. Repeating this pattern indefinitely, we get an arbitrarily long input sequence in which $\mathcal{S}_1$ loses $B_2 - B_1$ slices out of every batch, while $\mathcal{S}_2$ doesn't lose anything.

3.4.2 The penalty for variable slice sizes

We now go back to the case of variable slice size. The main difficulty is that since the slices can have different sizes, it may be the case that a server overflow occurs even when the server's buffer is not full. However, we have the following simple property, relating the throughput of the server in this case to the throughput of a server that is allowed to break slices arbitrarily.

Lemma 3.7 *Let $\mathcal{B}$ be any input stream, and let $\mathcal{B}'$ be the input stream derived from $\mathcal{B}$ by replacing each slice s by $|s|$ slices of size 1. Let $\mathcal{S}$ be the schedule obtained from the generic algorithm for $\mathcal{B}$ with server buffer size B, and let $\mathcal{S}'$ be the schedule of the generic algorithm for $\mathcal{B}'$ with server buffer size $B - L_{\max} + 1$. Then the throughput of the server's buffer in $\mathcal{S}$ is larger than or equal to the throughput of the server's buffer in $\mathcal{S}'$.*

Proof. Let $\mathsf{B}(t)$ and $\mathsf{B}'(t)$ denote the server's buffer at time t under $\mathcal{S}$ and $\mathcal{S}'$, respectively. Similarly, let $\mathsf{S}(t)$ and $\mathsf{S}'(t)$ denote the set of bytes sent by $\mathcal{S}$ and $\mathcal{S}'$ at time t, respectively. To prove the lemma, it is sufficient to show that $|\mathsf{B}(t)| \geq |\mathsf{B}'(t)|$ for all t. We show a stronger result by induction on time: We show that for all t,

$$|\mathsf{B}'(t)| \;\leq\; |\mathsf{B}(t)| \;\leq\; |\mathsf{B}'(t)| + L_{\max} - 1 .$$

The basis is trivial: both buffers are empty. For the inductive step, assume that the invariant holds at time t and consider time $t+1$. By definition of the schedules, the number of bytes arriving at time $t+1$ is the same in $\mathcal{B}$ and $\mathcal{B}'$. We proceed by case analysis.

- If no bytes are dropped by $\mathcal{S}'$ then no bytes are dropped by $\mathcal{S}$. This is true because by the induction hypothesis, at time t we have that the number of free slots in the server's buffer under $\mathcal{S}$ is $B - |\mathsf{B}(t)| \geq (B - L_{\max} + 1) - |\mathsf{B}'(t)|$, i.e., it is at least the number of free slots in the server's buffer under $\mathcal{S}'$. Also by induction we have that $|\mathsf{B}(t+1)| \leq |\mathsf{B}'(t+1)| + L_{\max} - 1$, since the number of bytes sent by $\mathcal{S}$ is at least the number of bytes sent by $\mathcal{S}'$. To see that $|\mathsf{B}'(t+1)| \leq |\mathsf{B}(t+1)|$, note that the number of bytes sent by $\mathcal{S}$ at time $t+1$ may be larger than the number of bytes sent by $\mathcal{S}'$ at time $t+1$ only if $|\mathsf{B}'(t+1)| = 0$.
- If some bytes are dropped by $\mathcal{S}'$ and no bytes are dropped by $\mathcal{S}$, then since $|\mathsf{B}'(t)| \leq |\mathsf{B}(t)|$ we have that $|\mathsf{B}'(t+1)| \leq |\mathsf{B}(t+1)|$. Also, if $\mathcal{S}'$ dropped some bytes then $|\mathsf{B}'(t+1)| = B - L_{\max} + 1$, and hence $|\mathsf{B}(t+1)| \leq |\mathsf{B}'(t+1)| + L_{\max} - 1$ in this case.
- If bytes are dropped by both $\mathcal{S}$ and $\mathcal{S}'$ at time $t+1$, then clearly $|\mathsf{B}'(t+1)| = B - L_{\max} + 1$. Moreover, $|\mathsf{B}(t+1)| \geq B - L_{\max} + 1$ since otherwise, another complete slice would have entered the buffer in $\mathcal{S}$. $|\mathsf{B}(t+1)| \leq |\mathsf{B}'(t+1)| + L_{\max} - 1$ because $|\mathsf{B}(t+1)| \leq B$. □

Using the results for buffers with different sizes, we derive the important property of servers for variable-size slices.

Corollary 3.8 *The throughput of the server's algorithm is at least $(B - L_{\max} + 1)/B$ times the optimal throughput.*

Proof. Denote the throughput of the server for the given input by θ. Consider a server with buffer size $B - L_{\max} + 1$ that gets the input stream modified by having each byte as an independent slice. Let θ' be the best possible throughput for this scenario. By Lemma 3.7, $\theta \geq \theta'$. Next, consider the best possible throughput θ^* of a server with buffer space B presented with the given input stream where each byte is a slice. By Lemma 3.6, $\theta' \geq \theta^*(B - L_{\max} + 1)/B$. The result follows. □

So consider the system using the same client's this algorithm as described in Sect. 3.1.2, and assume now that the slices have variable sizes. Using the results above, we conclude with the following theorem, which generalizes Theorem 3.5

Theorem 3.9 *The throughput of the generic algorithm for an arbitrary input stream $\mathcal{B}$ with slice sizes in the range $[1, L_{\max}]$ is at least $\frac{B-L_{\max}+1}{B}$ times the best possible throughput for $\mathcal{B}$.*

Proof. Lemmas 3.3 and 3.4 hold without change for the case of different slice sizes. The theorem follows from Corollary 3.8. □

4 Competitive analysis for general local value functions

The results of Sect. 3 imply that the decision of which slices are dropped lies exclusively at the server. In this section we zoom in to the server, and focus on the question: which slices should be dropped? Thus, in this section, we ignore the client completely, and all we consider is a single limited-space buffer with fixed drain rate, and an arbitrary arrival stream.

Now, if all slices are identical, it doesn't make any difference which slices are dropped (only how many are dropped matters). However, in many cases, not all slices are equal. Therefore, to make the question slightly more realistic, we generalize our treatment to general local value functions, i.e., we address the case that different slices have different values, and the benefit of a real-time schedule is the sum of the values of the slices played.

Our main tool here is competitive analysis. In competitive analysis, one compares the performance of an on-line algorithm to the performance of an optimal off-line algorithm. In our case, the competitive analysis is done as follows. For each input stream $\mathcal{B}$, we compute the benefit of the on-line algorithm, denoted by $online(\mathcal{B})$, against the benefit of an optimal off-line algorithm, denoted by $opt(\mathcal{B})$. An optimal off-line algorithm knows all the input stream in advance, and based on the entire stream it finds an optimal schedule. The competitive ratio for an input stream $\mathcal{B}$ is the ratio of the two benefits. The competitive ratio of an on-line algorithm is the worst-case competitive ratio over all input steams. An on-line algorithm is called c-competitive, if for any input sequence $\mathcal{B}$, we have that $opt(\mathcal{B})/online(\mathcal{B}) \leq c$.

In this section, we present the following results. First, we consider a natural greedy algorithm, and prove that it is $4 \cdot \frac{B}{B-2(L_{\max}-1)}$ -competitive for slices with sizes in the range $[1, L_{\max}]$ (in particular, this means that the algorithm is 4-competitive for unit-size slices). We also show that the greedy algorithm cannot be better than $(2 - \frac{2}{B})$ -competitive, even for fixed size slices. We then prove a lower bound of 1.23 on the competitive ratio of any deterministic on-line algorithm. We require the on-line scheduler to adhere to FIFO scheduling: it is clear that there exists an optimal off-line schedule which maintains FIFO order; also, this is a very natural restriction when discussing real-time smoothing.

For our algorithm, we choose a buffer size such that $B = DR$. Theorems 3.5 and 3.9, as well as Lemmas 3.3 and 3.4, guarantee that if $B = DR$ then no overflow or underflow will occur at the client buffer, i.e., no packet will arrive too early or too late. This allows us to restrict attention to the server buffer and the slices dropped from it, since any other slice will reach the client and will be played back on time.

4.1 The greedy algorithm for a single buffer

When we have a general local value function, it seems natural to drop slices with low value in exchange for slices with high values. However, when slices have different sizes, we should take their size into account too. To do that, we calculate the *byte value* for each slice: the byte value of slice s is $\frac{w(s)}{|s|}$. Using the byte value concept, we define the following rule of *greedy schedules*:

> *When a server overflow occurs: Discard the slices with the lowest byte value one by one in increasing byte value order, until the total size of remaining slices does not exceed B.*

Ties are resolved arbitrarily; for the analysis below, we fix a greedy algorithm and an arrival sequence. We emphasize that no slice is ever discarded once it starts being transmitted.

We now analyze the competitive ratio of the greedy algorithm. We treat the general case, where each slice can be of size $[1, L_{\max}]$. First we prove an upper bound of $4\frac{B}{B-2L_{\max}+2}$, and then a lower bound of nearly 2.

Theorem 4.1 *The competitive ratio of the greedy algorithm is at most $\frac{4B}{B-2L_{\max}+2}$.*

Let us give some intuition for the argument underlying the proof of Theorem 4.1. Consider unit size slices: in this case, the theorem states that the greedy algorithm is 4-competitive. To prove this, divide time into intervals of length D, and focus, in each interval, on the most valuable bytes that arrive. On the one hand, the optimal algorithm cannot accept more than the $2B$ most valuable bytes in each interval: at most B can be sent and at most B can be stored in the buffer. On the other hand, it is quite easy to see that the greedy algorithm sends, in the current and the following interval combined, bytes whose total value is at least as much as the value of the B most valuable bytes that arrive in the current interval. These two observations, put together, bound from below the total value sent by the greedy algorithm by a quarter of the best possible total value. Some care has to be exerted when extending the argument to the general case of variable-size slices (which also involves applying Theorem 3.9).

Let us now give a formal proof. First, some notation. For any set of slices A, let $w(A) \stackrel{\text{def}}{=} \sum_{s \in A} w(s)$. Also recall that the size of a set of slices is defined to be the sum of the slice sizes.

The following concept is central to our proof: For any set of slices F, we use $V(F)$ to denote the subset of F with maximal

value among all subsets whose size is at most $B - L_{\max} + 1$. More formally:

Definition 4.2 *Let $\mathcal{W}$ be the set of all subsets of F with size at most $B - L_{\max} + 1$. Then $V(F)$ is a set $F' \in \mathcal{W}$ such that for all $F'' \in \mathcal{W}$ we have $w(F'') \leq w(F')$. Ties are resolved by the way the greedy algorithm resolves them.*

In other words, $V(F)$ is the set of slices that would have resulted from the greedy rule above, when required to select slices from F, so that the total size selected is no more than $B - L_{\max} + 1$.

Note that $|V(F)| \geq \min(|F|, B - 2(L_{\max} - 1))$, because slice size is at most $L_{\max}$ by definition.

In the analysis below, it is convenient to assign a value to each byte: a byte b in slice s is assumed to have value $w(b) \stackrel{\text{def}}{=} \frac{w(s)}{|s|}$. Since the algorithm never preempts a slice after it started transmitting it, we have that the total benefit of the algorithm can be computed by summing the values of bytes the server sends.

We start with a lemma that says that the value of the most valuable $B - L_{\max} - 1$ bytes arriving in any interval is either sent during that interval or stored in the buffer when the interval ends.

Lemma 4.3 *Under the greedy algorithm, for any time t and duration $\ell \geq 0$, for the interval $I = [t, t + \ell - 1]$ it holds that*

$$w(\mathsf{S}(I)) \geq w(V(\mathsf{A}(I))) - w(\mathsf{B}_s(t + \ell)) \; .$$

Proof. Define the set $Y = V(\mathsf{B}_s(t) \cup \mathsf{A}(I))$. We claim that all slices in Y are either sent during I or stored in Greedy's buffer by the end of I. Formally:

$$Y \subseteq \mathsf{B}_s(t + \ell) \cup \mathsf{S}(I) \; . \tag{8}$$

Suppose that Eq. (8) is false. Then there exists a time in I in which a slice from Y is dropped. Let t_0 be the first such time, and b a byte from Y that was dropped at time t_0. By the rule of the greedy algorithm, we have that $w(b) < w(b')$ for all bytes $b' \in \mathsf{B}_s(t_0)$ with the possible exception of bytes of the slice transmitted at time t_0. This means that there are bytes in $B_s(t_0)$ of total size at least $B - L_{\max} + 1$ with value higher than $w(b)$. Since $\mathsf{B}_s(t_0) \subseteq \mathsf{B}_s(t) \cup \mathsf{S}(I)$, this is a contradiction to the definition of Y. We can therefore use Eq. (8) to conclude that

$$w(\mathsf{S}(I)) \; + w(\mathsf{B}_s(t + \ell)) \geq \; w(Y) \; ,$$

and the result follows, since $w(Y) \geq w(V(\mathsf{A}(I)))$ by definition. □

The next lemma says that the value stored in the buffer is a lower bound on the total value transmitted by Greedy in the following D time steps.

Lemma 4.4 *In the greedy algorithm, for any time step t,*

$$w(\mathsf{B}_s(t)) \leq \sum_{i=0}^{D-1} w(\mathsf{S}(t + i)) \; .$$

Proof. For $i = 0, \ldots, D - 1$, let $X_t(i)$ be the set of bytes transmitted in the time interval $[t, t+i]$ and the oldest (closest to being transmitted) $B - iR$ bytes in $\mathsf{B}_s(t+i-1)$. Intuitively, X_t is a window into the input stream that advances at the rate of transmission, so that it demarcates a fixed set of "advancing slots." Initially, X_t is exactly the set of bytes held in the buffer, and eventually, X_t is exactly the set of bytes transmitted in the last D time units. Formally, $X_t(0) = \mathsf{B}_s(t - 1)$ and $X_t(D - 1) = \cup_{i=0}^{D-1} \mathsf{S}(t + i)$.

We claim that for all $0 \leq i < D-1, w(X_t(i)) \leq w(X_t(i+1))$. To see that, consider time $t + i + 1$. First a set of slices $\mathsf{A}(t+i+1)$ arrive. Then a subset of $\mathsf{B}_s(t+i) \cup \mathsf{A}(t+i+1)$ may be dropped. In this case, by definition of the greedy algorithm, each byte is replaced by a byte whose value is not smaller, and hence, the value of $X_t(i)$ does not decrease. Finally, R bytes from the head of the buffer are transmitted, but the definition of $X_t(i + 1)$ makes sure that this does not affect the set of bytes we are considering. It follows from transitivity that

$$w(\mathsf{B}_s(t)) = w(X_t(0)) \leq w(X_t(D-1)) = \sum_{i=0}^{D-1} w(\mathsf{S}(t+i)) \; . \quad \square$$

Combining Lemma 4.3 with $\ell = D$ and Lemma 4.4 we have the following corollary.

Lemma 4.5 *In the greedy algorithm, for any interval $I = [t, t + D - 1]$,*

$$2w(\mathsf{S}(I)) \; + w(\mathsf{B}_s(t + D)) - w(\mathsf{B}_s(t)) \geq \; w(V(\mathsf{A}(I))) \; .$$

On the other hand, the following simple observation relates the optimal benefit to the value of $V(\mathsf{A}(I))$ for any interval I whose length is ℓ. The idea is that in the extreme case, the optimal schedule can collect the value of the $\ell R + B$ most valuable bytes that arrive in I.

Lemma 4.6 *Let I be any time interval of $\ell \geq 0$ steps. Then the total value of slices that arrive during I and can ever be sent by any algorithm with buffer space B is at most $\frac{\ell R + B}{B - 2(L_{\max} - 1)} w(V(\mathsf{A}(I)))$.*

Proof. Assume first that the number of all bytes arriving in I is less than $B - L_{\max} + 1$. In this case, $V(\mathsf{A}(I))$ is exactly the set of all arriving bytes, and the lemma clearly holds. Assume now that at least $B - L_{\max} + 1$ bytes arrive in I. Note that in this case, $|V(\mathsf{A}(I))| \geq B - 2(L_{\max} - 1)$. We will use this fact below. But first observe that no algorithm with buffer space B can accept (i.e., not drop) more than $B + \ell R$ bytes in any interval of ℓ time steps (that's the "leaky bucket" nature of the buffer). Since the bytes in $V(\mathsf{A}(I))$ have the greatest value among all bytes arriving in I, it follows that the value of any additional byte accepted by any algorithm is at most $\frac{w(V(\mathsf{A}(I)))}{|V(\mathsf{A}(I))|} \leq \frac{w(V(\mathsf{A}(I)))}{B - 2(L_{\max} - 1)}$. Hence, the total value any algorithm can accept in I using B space is at most $(B + \ell R) \cdot \frac{w(V(\mathsf{A}(I)))}{B - 2(L_{\max} - 1)}$. □

Using Lemma 4.6, and Lemma 4.5 with $\ell = D$ (and hence $\ell R = B$) we can now prove the theorem.

Proof of Theorem 4.1. Let T denote the last time step of the greedy schedule. Without loss of generality, we may assume that $\mathsf{B}_s(T + 1) = \emptyset$. Divide time into intervals $\{I_j\}_j$ of length D (the last interval may be shorter than D). Summing over all

intervals I_j, we get from Lemma 4.5 that

$$\sum_j w(V(\mathsf{A}(I_j))) \leq w(\mathsf{B}_s(T+1)) - w(\mathsf{B}_s(0)) + 2\sum_{t=0}^{T} w(\mathsf{S}(t))$$
$$= 2\sum_{t=0}^{T} w(\mathsf{S}(t)) \,, \qquad (9)$$

by our assumption that $w(\mathsf{B}_s(0)) = w(B_s(T+1)) = 0$. Eq. (9) means that the total benefit of the greedy algorithm is at least half of the sum of the $w(V(\mathsf{A}(I_j)))$. On the other hand, using Lemma 4.6 with $\ell = D$ and summing over all intervals, we get that the total benefit of the optimal algorithm is at most $\frac{2B}{B-2(L_{\max}-1)} \sum_j w(V(\mathsf{A}(I_j)))$. Combining with Eq. (9) completes the proof. □

Next we show that 2 is a lower bound on the competitive ratio of the greedy algorithm.

Theorem 4.7 *There exists an input stream $\mathcal{B}$ of fixed size slices with byte values in the range $[1, \alpha]$, on which the value of the optimal schedule is at least $2 - \epsilon$ times the value of the greedy scheduling, for $\epsilon \geq \frac{2}{\alpha+1} + \frac{1}{B+1}$, where B is the buffer size.*

Proof. We demonstrate the theorem by providing a parametric example, where all slices have size 1 and the link rate $R = 1$. Consider the following input stream, starting at time 0.

- At time 0, $B + 1$ slices arrive, with value 1 each.
- In each of time steps $1, 2 \ldots, B$, a single slice of value α arrives.
- Finally, at time $B+1$, $B+1$ slices of value α each arrive.

The greedy algorithm does not drop any slice at steps $0, \ldots, B$, since there is no overflow in these steps. Thus, after step B, the buffer contains B slices of value α each, and hence, at step $B+1$, B slices of value α are necessarily dropped by the greedy algorithm (one slice is sent out and B can be retained in the buffer). The overall benefit of the greedy algorithm for the given scenario is $1 \cdot (B + 1) + \alpha \cdot (B + 1)$. The optimal (off-line) algorithm for this case is as follows. In the first step, it drops all but one of the first value-1 slices, which is immediately sent out. At each steps $1, 2, \ldots, B$, the optimal algorithm sends a slice of value α. It therefore has an empty buffer after B steps, so it does not lose any slice of value α at step $B + 1$, resulting in total benefit of $1 + \alpha(2B + 1)$. Therefore, the competitive ratio is at least

$$\frac{(2B+1)\alpha+1}{(B+1)(\alpha+1)} = 2 - \frac{2B+\alpha+1}{(B+1)(\alpha+1)}$$
$$= 2 - \left(\frac{2}{(1+\frac{1}{B})(\alpha+1)} + \frac{1}{B+1}\right)$$
$$\geq 2 - \left(\frac{2}{\alpha+1} + \frac{1}{B+1}\right) \,,$$

since $B \geq 0$. □

4.2 A lower bound for deterministic on-line algorithms for a single buffer

It turns out that no deterministic on-line algorithm can be very close to optimal, as we show next. Our proof uses the assumption that on-line algorithms must send the slices in FIFO order, but it holds even when all slices have the same size.

Theorem 4.8 *For any deterministic on-line algorithm $\mathbf{A}$ and for a sufficiently large buffer size B, there exists an input sequence $\mathcal{B}$ such that the total value of an optimal schedule of $\mathcal{B}$ is at least 1.2287 times the total value of the on-line algorithm $\mathbf{A}$, i.e., $opt(\mathcal{B})/\mathbf{A}(\mathcal{B}) \geq 1.2287$.*

Proof. Assume that the link rate is $R = 1$, and all slices have unit size. We consider two scenarios, generalizing the argument of Theorem 4.7 to an arbitrary deterministic on-line algorithm. In both scenarios, the arrival sequence starts with $B + 1$ slices of value 1 arriving at time 0. In each of the next time steps, a single slice of value α arrives. Let t_1 be the last time that $\mathbf{A}$ sends a unit-size slice under this input sequence.

In the first scenario, the input stream ends at time t_1. The benefit of $\mathbf{A}$ is at most $1 \cdot (t_1+1)+\alpha \cdot t_1$. The optimal schedule would not drop any slice in this case, for a total benefit of $1 \cdot (B + 1) + \alpha \cdot t_1$.

In the second scenario, at time $t_1 + 1$ a burst of $B + 1$ slices of value α arrives. The benefit of $\mathbf{A}$ in this case is at most $1 \cdot (t_1 + 1) + \alpha \cdot (B + 1)$, while the optimal schedule in this case would drop all but the first 1-value slices and will not lose any α-value slice, for a total benefit of $1 + \alpha(t_1 + B + 1)$.

For $\alpha = 2$ and large B value, the ratio is $\min\left(\frac{z+2}{3}, \frac{2(1+z)}{1+2z}\right)$, where $z = B/t_1$. Equating both expressions and solving the resulting quadratic equation, we get that the minimum value is approximately 1.2287 (for $z \approx 1.6861$). □

Remark. Lotker, and independently Sviridenko (personal communications), have recently improved the analysis above by choosing $\alpha \approx 4.015$ and obtained a lower bound of 1.28197.

5 Experimental results

In this section we present some experimental results for video smoothing of clips obtained from the CNN archive [1]. The results reported in this section reflect typical values for these clips. We remark that the results in this section do not represent a comprehensive study of lossy smoothing: they are only intended to give some sense beyond the theoretical results of Sect. 4.

To get a better sense of the performance, in some cases we plot the weighted *loss* instead of throughput: weighted loss is the difference between the weight of the offered stream and the weight of the played stream. (We remark that the competitive ratio of weighted loss can be made arbitrarily large by using the bad scenarios of Theorems 4.7 and 4.8 and large α values.)

We tested the weighted loss in our simple model: given an MPEG stream, we assigned values to slices based on the type of frame they are a part of. Specifically, we assigned values of $12 : 8 : 1$ for $I : P : B$ frames, respectively. The weighted loss is the sum of the values of all slices lost in a given schedule, divided by the the sum of the values of all slices in the input.

We compared two simple algorithms. The first is the *FIFO* algorithm (a.k.a. *Tail-Drop* algorithm), if an overflow occurs at time i, then slices from frame i are discarded (intuitively, all

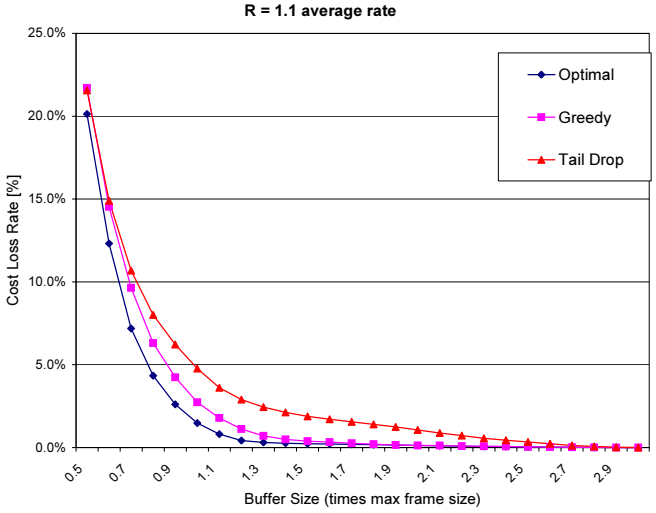

Fig. 2. The weighted loss of the Tail-drop and Greedy algorithms using a link rate 10% above the global average rate of the offered stream. Each byte is a slice

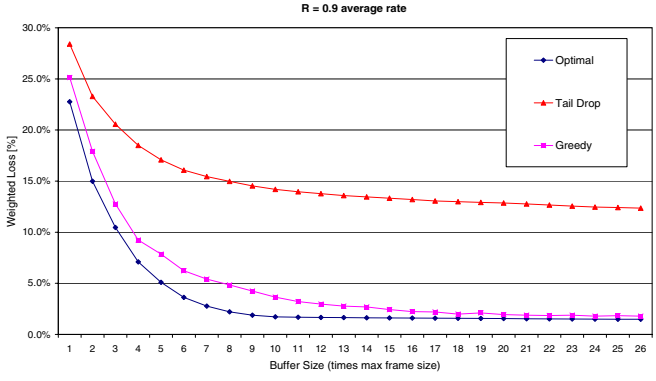

Fig. 3. Fraction of lost value in a high quality clip of the Tail-drop and Greedy algorithms under link rate 10% below the average rate. Each byte is a slice

overflow is from the "tail" of the server's buffer). The second algorithm is the *Greedy* algorithm, in which whenever an overflow occurs, the slices with the lowest value are discarded from the buffer, regardless of their location in the server's buffer or in the current input frame. We looked at two extremes for the slice size: on one extreme, each *byte* is an individual slice; and on the other extreme, each *frame* is an individual slice. For comparison purposes, we also plot the optimal weighed loss possible (by an off-line algorithm) for the given buffer size and link rates.

The data used in our experiments has the following characteristics. The average frame size is about 38 KBytes, and the maximum frame size is about 120 KBytes. The frequencies of I, P, and B frames are about 8%, 31%, and 61%, respectively.

5.1 Single byte slices

Clearly, both Tail-Drop and Greedy perform the same if no (or very little) loss occurs. In Fig. 2 we see a comparison of the total value the algorithms lost due to overflow, for a high-quality input stream, in the single-byte slices model, where

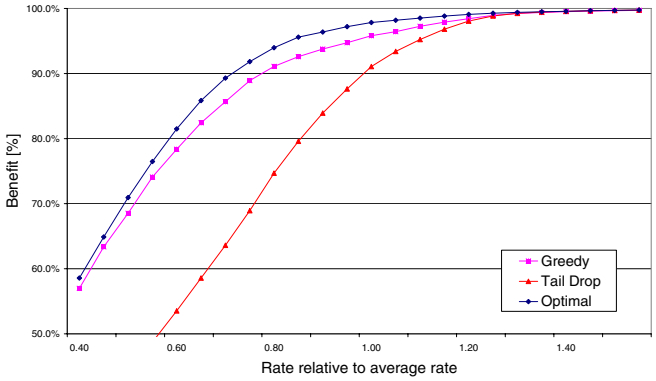

Fig. 4. Benefit of Tail-Drop, Greedy and Optimal relative to the total benefit under varying link rate. Each byte is a slice

the link rate is 10% above the average stream rate (the average stream rate is defined to be the total number of bytes in the stream divided by the total number of frames in the stream). As expected, the Greedy algorithm outperforms the Tail-Drop algorithm in all cases, and the overall weighted loss is never too high, assuming that the buffer size is more than the size of the largest frame.

In Fig. 3 we consider the same setting with a link whose speed is set to be 10% below the average rate. Clearly, in this case, at least 10% of the bytes are lost (ignoring one full buffer's worth). However, the *weighted* loss may be significantly lower, as is the case for the Greedy algorithm and the optimal schedule. On the other hand, the weighted loss of the Tail-Drop algorithm is above 10%. This phenomenon can be explained by the fact that in MPEG streams, the valuable bytes come in large bursts; since Tail-Drop loses part of the incoming burst, its weighted loss exceeds its unweighted loss.

5.2 Influence of the link rate

In Fig. 4, we plot the benefit of the Greedy, Tail-Drop and the optimal schedule, as we vary the link rate. It can be seen that the Greedy algorithm manages to salvage most of the benefit even when the rate drops well below the average rate. One possible interpretation is that the Greedy algorithm saves link bandwidth by investing more processing in the server.

5.3 Whole frame slices

We also consider the case where each slice is a complete frame. This case is very different: In Fig. 5, we compare the *optimal* weighted loss for whole-frame slices with the *optimal* weighted loss for single-byte slices. The difference in the weighted loss may be as high as nearly a factor of 4 when the buffer is very small, but it shrinks when the buffer size increases.

Figure 6 compares the weighted loss of the Tail-Drop and Greedy algorithms for different slice sizes. It turns out that the large difference in favor of the Greedy algorithm that exists in byte-size slices is only partially preserved in the whole-frame slice case. However, in this case too, the Greedy algorithm always does better than the Tail-Drop algorithm, especially for moderate size buffer.

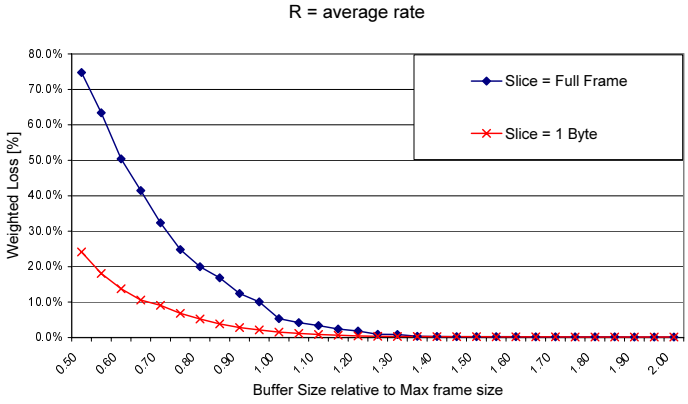

Fig. 5. The optimal weighted loss as a function of the buffer size, for single-byte slices and whole-frame slices

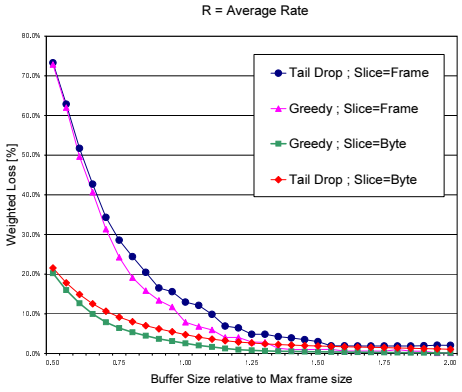

Fig. 6. The weighted loss of Tail-Drop and Greedy as a function of the buffer size, for single-byte slices and whole-frame slices

6 Conclusion

In this paper we investigated a simplified version of the lossy smoothing model, where the goal is to minimize the number of lost slices. In particular, we showed that for any given delay requirement and bandwidth limit, the least possible number of slices dropped can be attained when the buffer sizes at the client and the server equal the delay-bandwidth product, and server sends out the data as quickly as possible. We showed that if any two of the three parameters (buffer space, delay, and bandwidth) are fixed, then setting the third parameter differently than what our formula implies will result in excessive data loss, or in resource wastage. We then studied a more refined model where each slice has an intrinsic "value," and the goal of the smoothing schedule is to maximize the total value of slices played out on time. We applied competitive analysis to this model, and proved that the natural Greedy algorithm is competitive. This result is supported by a few simulations using traces of MPEG streams.

Our analytical results hold for variable slice sizes, but for fixed link delays. We justified this restriction by assuming that some jitter control algorithm is employed. However, such an algorithm adds to the buffer space requirement and to overall delay. The obvious question that arises is whether one can prove similar results for links with positive jitter. We leave this question open. We also leave open the issue of more "proactive" algorithms for overflows (as opposed to the greedy algorithm, that never preempts a packet before an overflow occurs).

In general, we view this paper as a first step in the study of lossy smoothing. Being a first step, it makes many serious simplifications. Yet, we believe that our results are meaningful in the sense that they set some "rules of thumb" regarding resource allocation for smoothing, and they provide some basic techniques for analyzing losses for FIFO streams with bounded buffers. We hope that more steps will follow.

Acknowledgements. The authors thanks the anonymous reviewers for their helpful comments.

References

1. www.nmis.org/NewsInteractive/CNN/Newsroom
2. MPEG-1 standard (ISO/IEC 11172), 1992
3. MPEG-2 standard (ISO/IEC DIS 13818), 1994
4. Borodin A, El-Yaniv R: Online computation and competitive analysis. Cambridge University Press, ♣ 1998
5. Chang R-I, Chen M-C, Ho J-M, Ko M-T: An effective and efficient traffic smoothing scheme for delivery of online VBR media streams. In: Proceedings of IEEE INFOCOM, 1999
6. Civanlar M, Cash G, Haskell B: RTP payload format for bundled MPEG, May 1998. Internet RFC 2343.
7. Duffield NG, Ramakrishnan KK, Reibman AR: SAVE: An algorithm for smoothed adaptive video over explicit rate networks. IEEE/ACM Transactions on Networking 6(6):717–728 (1998)
8. Feng W, Rexford J: Performance evaluation of smoothing algorithms for transmitting prerecorded variable-bit-rate video. IEEE Trans. on Multimedia 1(3):302–313 (1999)
9. Grosslauser M, Keshav S, Tse DNC: RCBR: A simple and efficient service for multiple time-scale traffic. IEEE/ACM Transactions on Networking 5(6):741–755 (1997)
10. Ni TYJ, Tsang D: A CBR transport technique for MPEG-2 video-on-demand connections over ATM networks. In: Proc. IEEE ICC 96, pp 1391–1395. June 1996
11. Jiang Z, Kleinrock L: A general optimal smoothing video algorithm. In: Proc. IEEE INFOCOM. Mar. 1999
12. Keshav S: An engineering approach to computer networking. Addison-Wesley Publishing Co., ♣ 1997
13. Lam SS, Chow S, Yau DKY: An algorithm for lossless smoothing of MPEG video. IEEE/ACM Transactions on Networking 4(5):697–708 (1996)
14. Rexford J, Sen S, Dey J, Feng W, Kurose J, Stankovic J, Towsley D: Online smoothing of live, variable-bit-rate video. In: Proc. International Workshop on Network and Operating Systems Support for Digital Audio and Video, pp 249–257. May 1997
15. Rexford J, Towsley D: Smoothing variable-bit-rate video in an internetwork. IEEE/ACM Transactions on Networking, pp 202–215. Apr. 1999
16. Salehi J, Zhang Z, Kurose J, Towsley D: Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing. IEEE/ACM Transactions on Networking 6(4):397–410 (1998)
17. Sen S, Dey J, Kurose J, Stankovic J, Towsley D: CBR transmission of VBR stored video. In: SPIE Symposium on Voice Video and Data Communications, Nov. 1997
18. Sen S, Rexford J, Towsley D: Proxy prefix caching for multimedia streams. In: Proc. IEEE INFOCOM, Mar. 1999
19. The ATM Forum Technical Committee. Traffic management specification version 4.0, Apr. 1996. Available from www.atmforum.com

20. Wrege DE, Knightly EW, Zhang H, Liebeherr J: Deterministic delay bounds for VBR video in packet-switching networks: fundamental limits and practical trade-offs. IEEE/ACM Transactions on Networking 4(3):352–362 (1996)
21. Zhang H: Service disciplines for guaranteed performance service in packet-switched networks. Proceedings of the IEEE 83(10): ♣–♣ (1995)
22. Zhang Z-L, Nelakuditi S, Aggarwal R, Tsang RP: Efficient selective frame discard algorithms for stored video delivery across resource constrained networks. In: Proc. IEEE INFOCOM, Mar. 1999
23. Zhao W, Seth T, Kim M, Willebeek-LeMair M: Optimal bandwidth/delay tradeoff for feasible-region-based scalable multimedia scheduling. In: Proc. IEEE INFOCOM 98, 1998