ACM SIGACT News Distributed Computing Column 4

Sergio Rajsbaum*

July 27, 2001

Abstract

The Distributed Computing Column covers the theory of systems that are composed of a number of interacting computing elements. These include problems of communication and networking, databases, distributed shared memory, multiprocessor architectures, operating systems, verification, internet, and the web.

This issue consists of a contribution by Boaz Patt-Shamir.

Request for Collaborations: Please send me any suggestions for material I should be including in this column, including news and communications, open problems, and authors willing to write a guest column or to review an event related to theory of distributed computing.

Transmission of Real-Time Streams

Boaz Patt-Shamir [1]

1 Introduction

Communication in general can be defined as the reproduction of a given sequence of bits in a remote location. E-mail and FTP are popular examples for general data communication services over the Internet. In *interactive* communication, we usually require that communication is done sufficiently quickly so as not to annoy a human user. This means that there are constraints on the time that may elapse since a bit is input at the sender's side until it is output on the receiver's side. Telephone is an example for this type of communication. *Real-time* communication is somewhere in the middle between these extremes: in real-time streams, the requirement is that the *relative* timing of bits is constrained, i.e., the absolute delay of each bit is not very important, so long as the time interval between any two bits at the receiver is the same at the receiver. Keeping the relative timing allows the receiver to precisely reconstruct the stream generated by the sender. Video-on-demand is an example for real-time communication: the user is usually willing to tolerate some delay before the movie starts playing, but once the movie is running, it should run with its original timing.

One of the main difficulties in real-time communication is that often, the bit rate of a real-time stream varies with time (possibly due to compression), while communication bandwidth is fixed. This fundamental conflict has many possible solutions:

*Compaq Cambridge Research Laboratory, One Cambridge Center, Cambridge, MA 02142-1612. Sergio.Rajsbaum@compaq.com, rajsbaum@math.unam.mx. On leave from Instituto de Matemáticas, UNAM.

[1] boaz@eng.tau.ac.il. Dept. of Electrical Engineering, Tel-Aviv University, Tel-Aviv 69978, Israel.

- The *conservative rich* approach is simply to reserve the maximal possible bandwidth required by the stream. This approach is unavoidable if minimum delay is the key concern, and no information loss is allowed [6]. In many scenarios, however, the resulting cost is huge: in an encoded video stream, for example, the peak bandwidth is often 10 times larger than the average bandwidth. Audio compact disks use this kind of approach.

- The *conservative poor* approach is to adjust the stream to the available bandwidth by truncating some of the information. This approach can be used only when graceful degradation of service is allowed, e.g., when transmitting voice or images, and different levels of compression can be applied [3].

- Another approach, called *statistical multiplexing*, is to let the constant bit-rate channel serve many concurrent streams, and to rely on the hope that not all peak rates will occur simultaneously. This approach can be justified analytically under some statistical conditions which may not be true in general. In fact, statistical multiplexing seems more questionable with the discovery that computer communication exhibits self-similar nature and thus does not have finite variance [7].

- The most natural approach in case that absolute delay constraints are not very restrictive, though, is the idea of *smoothing*, which is the focus of this article.

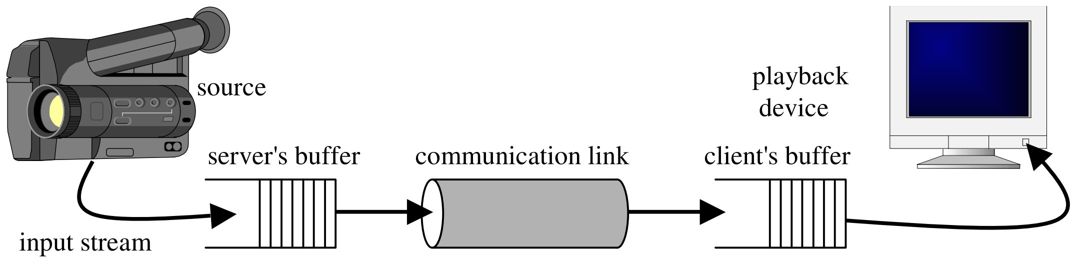

Figure 1: A schematic representation of a smoothing system. The link is lossless.

In smoothing, the basic idea is to trade bandwidth for space and latency. This is done as follows (see Figure 1). The bits of the input stream generated by the source are not transmitted directly over the communication link: first they are stored in a *server's buffer*; the server submits bits stored in its buffer to the link when it deems appropriate, subject to the link rate constraint. Bits arriving at the other side of the communication link are first stored in a *client's buffer*, which delivers them to the play-out device after a reconstruction action. The added flexibility provided by the buffers is used to create a 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 [1, 2]. 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. (Typical numbers for MPEG news video are reducing peak bandwidth by a factor of 2-10, using 1-5 megabytes of memory, with queuing delay of less than 5 seconds.)

The basic problem in smoothing is determining what is the link rate, buffer sizes, the play-out delay, and, of course, what is the right algorithm to use. There are a few variants that may be considered for the smoothing problem: in the *off-line* variant, we assume that the stream is fully given in advance (as is the case with stored video), and in the *on-line* variant, the algorithm knows about the input only when it arrives (e.g., news broadcast). Another aspect of interest is whether the system must be *lossless* or can it be *lossy*, i.e., must the input stream be reproduced precisely at the play-out device, or is the smoothing system allowed to drop some of the information. In the case of lossy smoothing, one needs to have some measure of the quality of the stream that is played out eventually.

In the remainder of this short article we'll survey some of the interesting ideas behind lossless off-line smoothing and lossy on-line smoothing.

Disclaimer. This article is *not* a comprehensive survey of the field. Rather, it tries to give a flavor of the ideas related to the problem of smoothing (most probably, with many unfrogivable omissions). The author humbly accepts all the blame, and welcomes all comments.

2 Formal Model

For simplicity, let us first consider bit streams as continuous flows. We describe the input stream by its *arrival function*, denoted $A(t)$, which tells us, for each time point t, how many bits have arrived by time t. For example, the number of bits that arrive in the time interval $(t, t+1]$ is $A(t+1) - A(t)$. For the off-line case, we shall assume that the arrival function is fully known in advance, and that all bits arrive in the time interval $[0, T]$ for some known T. Similarly, the *play-out function*, denoted $P(t)$, says how many bits were output (to the play-out device) by time t. A *transmission schedule* is also represented by a function $S(t)$ which says how many bits were transmitted from the server's buffer by time t. The *link rate* t is simply $\frac{dS}{dt}$.

To describe actual bit streams, we can use right-continuous step functions. Suppose that a stream is described by a function f (f can be the arrival, send, or playout function). In this case the rate of the stream at a given (discrete) time t is $f(t) - f(t-1)$, and it is more convenient to represent a stream as a sequence $(a_1, \ldots, a_T)$, where a_i is the number of new bits at time i, i.e., $a_i = f(i) - f(i-1)$.

We shall assume in this article that the only delays in the system are the queuing delays in the server's and client's buffers. This abstraction allows us to isolate the problem of smoothing from the (important) problem of jitter (variable delay).

3 Off-line Lossless Smoothing

In the off-line lossless smoothing scenario, we are given a full description of the input stream $A(t)$, and the play-out function $P(t)$ is completely characterized by the *play-out delay* D, in the sense that $P(t) = A(t-D)$ for all $t \geq D$.

Consider the geometrical representation of the streams (where the horizontal axis denotes time, and the vertical axis denote bit numbers): see Figure 2. The requirement of lossless real-time communication as we defined it is that the curve describing the output stream $P(t)$ will have the same shape as the curve describing the input stream $A(t)$, except for being shifted to the right: the amount of shifting to the right corresponds to the total delay each bit experiences from the time it is input by the source until it is output by the play-out device. Given a time point, the vertical distance between the input and the output curves corresponds to the number of bits currently buffered in the system (either in the server's buffer or the client's buffer).

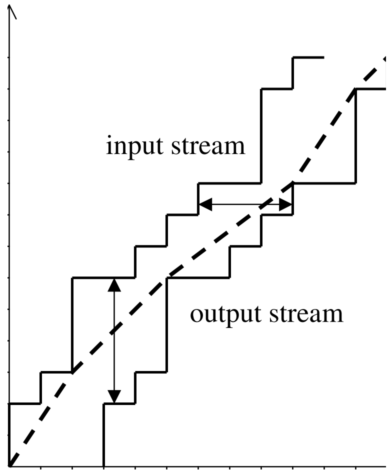

Figure 2: Geometrical representation of a smoothing schedule. A point (x, y) on a curve represents bit number y at time x. Horizontal distance represents delay, and vertical distance represents buffer space. The dashed curve represents a feasible lossless schedule.

With this representation, a lossless smoothing schedule is just any monotonically increasing function $S(t)$ that is never below the output curve and never above the input curve, i.e.,

$$P(t) \leq S(t) \leq A(t) \quad \text{for all } t. \tag{1}$$

The curve of $S(t)$ represents which bits are sent in each step: at any given time, bits above the curve are stored at the server's buffer, and bits below the curve are stored at the client's buffer. The slope of the schedule is the transmission speed. Staring at the curves, the reader may convince herself in the truthfulness of a few simple observations, such as:

> A stream can be smoothed to any desired link rate $R > 0$ if the play-out delay is not constrained: all we have to ensure is to have the play-out delay D large enough so that $\frac{s(t)}{t-D} \leq R$ for all $t \in [0, T]$.

It is also quite easy to answer one of our main questions: what is the minimum delay and the link rate required to transmit the stream if the rate is fixed, and the link is to be fully utilized? (A link with rate R is said to be *fully utilized* in a given time interval if during that interval it transmits R bits in each time unit.) We give a physical description of the algorithm, as it is quite intuitive. Considering the geometric representation, we can think of the input and output curves as rigid "walls" of the same shape (see Figure 3). We start the algorithm with these walls very far apart, and we slide the output curve horizontally, which corresponds to reducing the play-out delay; the only thing that stops the walls from meeting each other is an infinite length, zero width rigid straight wall between them which is not attached to anything. We push the output wall to the left as tightly as we can: Clearly, the resulting location of the free wall describes the schedule. Algorithmically, this problem can be solved using convex hull techniques.

We can also prove the following fact.

Theorem 1 *Let an input stream be given, and let R be the link rate. Let D be the minimal play-out delay D that keeps the link fully utilized. Then the buffer storage required at the client and the server is exactly $D \cdot R$.*

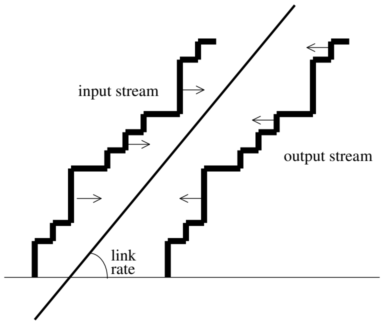

Figure 3: The wall-pushing algorithm. The arrival function and the play-out functions start very far apart, and a fixed rigid line of slope R is placed between them. The functions are pushed on the horizontal axis toward each other until they cannot be pushed further.

Proof: Let S be the optimal schedule, and consider the "wall pushing" algorithm described above that generates S. By the algorithm, there is (at least one) time point t where $S(t) = P(t)$: at time t the curve of the schedule touches the curve of the play-out stream, which means that the bit arriving at time t to the client is immediately played out. This bit must have waited in the server's buffer for D time units, during which DR bits were transmitted from the server's buffer (by the assumption that the link is fully utilized). Hence the size of the server's buffer is at least DR bits. On the other hand, since no bit is ever delayed more than D time units, it is clear that the server's buffer never contains more than DR bits, and therefore the server's buffer size is exactly DR. A dual argument applies to the clients buffer: There must exist a time point t' in which $A(t') = S(t')$, i.e., the bit arriving at time t' is immediately transmitted over the link. Since this bit waits D time units in the client's buffer, it follows that when it is played out, the client's buffer holds exactly DR bits, and hence its size is at least DR. Conversely, since no bit resides in the client's buffer more than D time units, and since the fill rate of the buffer is R time units, we have that the number of bits that enter the client buffer since any given bit arrives and until it is played out is at most DR, and therefore the size of the client's buffer is exactly DR. ∎

Salehi, Zhang, Kurose and Towsley [11] present a beautiful solution to the case where the rate can be piecewise constant. In this case the goal is to minimize the variability of the link rate and its maximum value. The measure used in [11] is *majorization*, which can be viewed as a generalization of the standard statistical variance measure [9]. Formally, majorization is defined as follows.

Definition 1 *Given a vector $A = (a_1, a_2, \ldots, a_n) \in \mathbb{R}^n$, let $(a_{[1]}, a_{[2]}, \ldots, a_{[n]})$ be the vector obtained from A by rearranging its coordinates in non-increasing value, i.e., $a_{[i]} \geq a_{[i+1]}$ for $1 \leq i < n$. We say that a vector $A = (a_1, a_2, \ldots, a_n)$ is* majorized *by a vector $B = (b_1, b_2, \ldots, b_n)$, denoted $A \prec B$, if for all $m < n$ we have $\sum_{i=1}^{m} a_{[i]} \leq \sum_{i=1}^{m} b_{[i]}$, and $\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} b_i$.*

Intuitively, majorization is an order among vectors with equal sum of coordinates, where vectors whose coordinates are more balanced are majorized by less-balanced vectors. For example, we have that $(2, 2, 2) \prec (1, 3, 2) \prec (4, 1, 1)$. In particular, the vector with all coordinates equal is majorized by any other vector with the same total sum of coordinates.

Consider majorization as a measure of smoothness of a schedule, where the schedule is given as a sequence of the number of bits transmitted in each step. Majorization has nice properties that coincide with some of our intuitive requirements. For example, it is straightforward to verify that if $A, B \in \mathbb{R}^n$ satisfy $A \prec B$, then:

- $\max\{a_1, \ldots, a_n\} \leq \max\{b_1, \ldots, b_n\}$, i.e., the maximal transmission rate in A is not larger than it is in B.
- $\sum_{i=1}^{n} a_i^2 \leq \sum_{i=1}^{n} b_i^2$, i.e., $\mathrm{Var}(A) \leq \mathrm{Var}(B)$, where Var is the statistical variance function.

More generally, it can be proved that $A \prec B$ if and only if $\sum_{i=1}^{n} f(a_i) \leq \sum_{i=1}^{n} f(b_i)$ for all continuous convex real functions f [9].

Salehi *et al.* give a polynomial-time algorithm for finding the best schedule, in the majorization sense, for any given stream, when either the play-out delay is given, or when the client buffer size is given. We will not give the detailed algorithm here, but just explain it intuitively. Consider once again the visualization of the stream curve as a rigid "wall" (see Figure 4). Given the delay (or buffer space) constraint, we have both the input and the output stream fixed. The question is now to find the best schedule. The idea of the algorithm is to try to find the curve obtained by pulling a *rubber band*, tied in one end to the origin (the point corresponding to the even of the first input arriving at the server), and in the other end, to the topmost point of the output curve (the point corresponding to the event of the last bit departing at the client). This piecewise linear function can proved to be the smoothest schedule with respect to the majorization order.

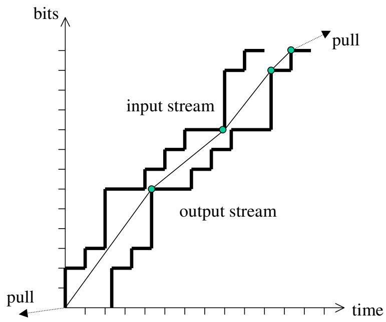

Figure 4: The rubber-band algorithm. The arrival function and the play-out functions are fixed. An elastic rubberband is placed between them and pulled outwards. The resulting shape is the best schedule w.r.t. majorization. The dots indicate rate changes.

The rubber-band algorithm can actually solve a more general problem, where the only connection between the input stream and the output stream is that they both contain the same total number of bits: this means that the walls need not be of the same (shifted) shape, but rather all they need to satisfy is that their topmost points have the same y coordinate. This observation is the starting point for another deep work of Rexford and Towsley [10]. In that work, the problem of determining a smoothing schedule for a single link is generalized to the scenario of a *line* of links, and the play-out function does not have necessarily the same shape as the arrival function (this scenario is justified in the context of multiple interconnected networks). It turns out that the line problem, where each intermediate node has its prescribed buffer space, can be reduced to the single link problem where the link has different input and output streams (with the same total number of bits). More specifically, in [10] they show that the optimal schedule for the line consists of optimal single-link schedules. And in the case of a single link with different input and output streams, Rexford and Towsley give efficient algorithms for finding the buffer space allocation and play-out delay that minimize the peak rate.

4 Lossy On-Line Smoothing

When the stream is not known in advance, it makes little sense to try to keep the smoothing schedule lossless, unless we are allowed to change the link rate: It can always be the case that the allocated bandwidth is not sufficient. Some work was done to study the way link rate should be changed to avoid data loss. In this section, we consider the scenario where the link rate is fixed, and consequently, some of the bits in the stream may be lost. We formalize this assumption by a simple additive model, where the value of a given stream is the sum of the value of its individual bits, where each bit has an intrinsic value assigned somehow. Clearly, this is a gross simplification: Firstly, the atomic unit of loss is usually much larger than a bit. For example, the Internet defines a basic units called *slice* in MPEG streams, and any set of dropped bits is an integral number of such slices [4]. Moreover, there is no agreed way of assigning a value to a slice: it seems quite subjective. However, our model applies to the case where all slices have the same size, and for *any* given slice-value function.

Bytes may be dropped due to the following restrictions.

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 number of bits arriving is larger than the current size of free space in the buffer, some of bytes (perhaps stored at the server) must be discarded.

Client Overflow: Similarly, if the number of bits arriving at the client is larger than its current number of free slots, then some bytes must be dropped.

Client Underflow: It may be the case that bits arrive at the client too late to be played out, and are therefore useless.

Early Drop: The algorithm controlling the schedule may drop bytes arbitrarily, possibly to avoid dropping bytes later.

Our way of measuring the performance of a smoothing system is based on competitive analysis [12]: given an input stream, we compare the total value played by the on-line algorithm with the best possible total value played by an off-line algorithm. The competitive factor of an algorithm is the maximal ratio over all input streams.

Determining the best schedule can be done under many circumstances:

- Given the client's and server's buffer size (denoted B_c and B_s, respectively) and the maximal link rate R. The question now is what is the minimal play-out delay, and what bits to drop?
- Given B_c and B_s as above, and maximal allowed play-out delay, what is the minimal peak rate R, and what bits to drop?

etc.

Algorithm for maximizing throughput. We start by considering the simplest scenario, where all bits have the same value. In this case, our goal is to play out the maximal possible number of bits in time. Suppose that the buffer spaces are equal to a given parameter B, that the link rate R is given, and the play-out delay is to be chosen by the algorithm. It turns out that a very simple solution maximizes the number of played bits, if the following simple connection is maintained between the buffer space, link rate, and play-out delay:

$$D = \frac{B}{R}\ . \tag{2}$$

The algorithm (called "generic" below) is as follows. The server's job is just to push the arriving bits out of the link as quickly as possible in FIFO order, tagged with their arrival time somehow (the timing can be relative if we ignore the propagation delay over the link). If there is no room for new bits in the server's buffer, then *an arbitrary set* of the available bits (which include the bits currently stored in the buffer, as well as the new bits) is discarded. We do require that bits that are transmitted, will be transmitted in the order of their arrival. The client's algorithm is even simpler: when the first bit arrives at the client's buffer, a *client's clock* is reset to 0. Each bit that arrives at the client's buffer tagged with arrival time t is re-tagged with label $t + D$, where $D = B/R$ is the play-out delay. A bit is played out when its tag equals the value of the client's clock.[2]

Let us now prove that the generic algorithm above loses the least number of bytes among all algorithms with buffer space B and link rate R. First note that the server transmits the largest possible number of bits among all servers with buffer space B: this can be shown by induction, using the "greedy" policy of the server, that always transmits as many bits as possible. Next, we prove that the clients does not drop any bit, neither by overflowing no by underflowing:

1. First, note that no bit ever waits at the server's buffer more that $B/R = D$ time units: at the time of arrival there are at most B bits ahead of it, and if the bits is not dropped, it will reach the head of the buffer in no more than B/R time units. It follows that no bits arrives at the client too late.

2. Next, note that that since no bit ever remains in the client's buffer more than D time units, there is no overflow in the client's buffer: at most $DR = B$ bits may be in the buffer when a bit is played out.

From the above analysis it is quite clear that the buffers at the server and the client should be of equal space, i.e., $B_s = B_c$. If they are of different size, damage might result from using any space more than $\min(B_s, B_c)$. To see that, consider for example the case of $B_s = 4$ and $B_c = 2$, and the link rate is 1. Suppose that the arrival stream is the following:

arrival time	# bits
1	4
2	2
3	2

Since the client's buffer size is 2, the system is bound to lose at least two of the bits that arrive at time 1: no other bits will be lost if we don't try to use the extra buffer space at the server, i.e., if we drop two of the first four bits at the server. However, if the server does not discard these two bits, then necessarily, some of the bits arriving at times 2 and 3 will be lost too.

Using similar arguments, it is not difficult to see that whenever two of the three parameters (buffer space, play-out delay, and link rate) are given, the optimal choice for the third one is given by Eq. (2). "Optimality" here is with respect to the number of bits lost.

[2] Recall that we assume that the only delay in the system is the queuing delay in the server's and client's buffer. Additional fixed delays can be incorporated too in a straightforward way.

Other cost functions. One can think of many possible context functions: the simplest generalization of our model is general additive cost functions, where each bit has an arbitrary value, and the value of a stream is the sum of its bits. Using the algorithm outlined above, the problem of delivering the maximal possible value reduces to a single buffer question, since the client's buffer never overflows nor underflows. The natural policy in this case is that whenever an overflow occurs (at the server's buffer), the server discards the lowest value bits. This simple greedy policy is in fact 2-competitive (i.e., it delivers at least half of the best possible benefit) [5], and the bound is tight [8]. The best lower bound for this problem, however, is about 1.28 [13].

5 Conclusion

We have seen in this brief article a few of the problems and possible solutions for real-time streams. Many interesting problems remain unsolved, of which we state but a few.

- What is the best way to deal with links with variable delay in a competitive fashion?
- What kind of cost functions truly model the "value" of the lossy transmission? What cost functions admit competitive soltions?

Other implied questions concern the management of a single buffer (whereas in smoothing we have two buffers).

References

[1] MPEG-1 standard (ISO/IEC 11172), 1992.

[2] MPEG-2 standard (ISO/IEC DIS 13818), 1994.

[3] N. G. Duffield, K. K. Ramakrishnan, and A. R. Reibman. SAVE: An algorithm for smoothed adaptive video over explicit rate networks. *IEEE/ACM Transactions on Networking*, 6(6):717–728, 1998.

[4] D. Hoffman, G. Fernando, V. Goyal, and M. Civanlar. RTP payload format for MPEG1/MPEG2 video. Internet RFC 2250, Jan. 1998.

[5] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in qos switches. In *Proc. 33rd Ann. ACM Symp. on Theory of Computing*, Jul. 2001.

[6] S. S. Lam, S. Chow, and D. K. Y. Yau. An algorithm for lossless smoothing of MPEG video. In *Proc. ACM SIGCOMM*, London, England, 1994.

[7] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Trans. Networking*, 2(1):1–15, Feb. 1994.

[8] Y. Mansour, B. Patt-Shamir, and O. Lapid. Optimal smoothing schedules for real-time streams. In *Proc. 19th ACM Symp. on Principles of Distributed Computing*, pages 21–29, Jul. 2000.

[9] A. W. Marshall and I. Olkin. *Inequalities: Theory of Majorization and its Applications.* Academic Press, 1979.

[10] J. Rexford and D. Towsley. Smoothing variable-bit-rate video in an internetwork. *IEEE/ACM Transactions on Networking*, pages 202–215, Apr. 1999.

[11] J. Salehi, Z. Zhang, J. Kurose, and D. Towsley. Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing. *IEEE/ACM Transactions on Networking*, 6(4):397–410, Aug. 1998.

[12] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. of the ACM*, 28(2):202–208, 1985.

[13] M. Sviridenko, 2001. Personal Communication.