Stop-and-Go Service Using Hierarchical Round Robin

S. Keshav

AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA
keshav@research.att.com

Abstract

The Stop-and-Go service discipline allows a rate-controlled Virtual Circuit to obtain (small) delay-jitter bounds independent of the number of hops along its path. This may prove to be desirable for isochronous applications in wide-area virtual circuit networks. We consider how to integrate the Stop-and-Go and Hierarchical Round Robin service disciplines to allow delay-jitter bounded communication in large homoge-

1. Introduction

It has been argued that controlling the variation in the end-to-end delay received by a rate-controlled virtual circuit would be a useful service in future networks (Golestani, 90; Verma, Zhang and Ferrari, 91). This variation is usually measured by the delay jitter, defined as the difference between the maximum and minimum possible end-to-end delay received by any data unit. This concept is graphically illustrated by Figure 1, which shows the histogram of delays received by cells on circuits A and B. We see that both circuits have the same delay bound, but circuit B has a tighter delay-jitter bound.

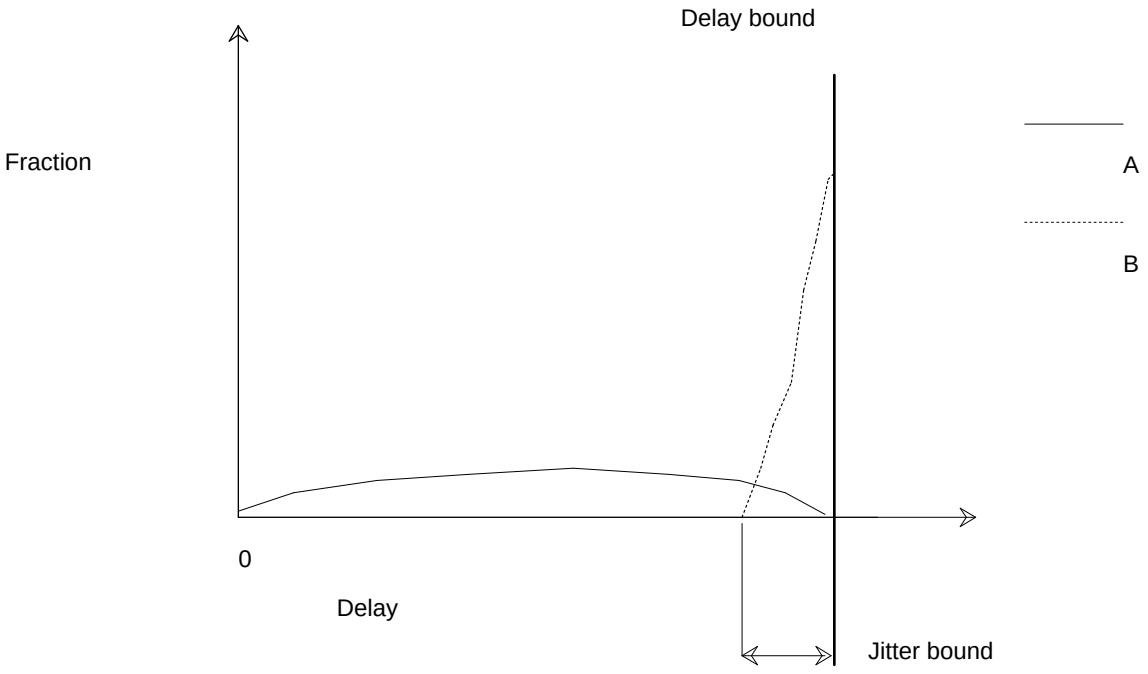

Figure 1: Delay and delay-jitter bounds

Recently, Golestani proposed the Stop-and-Go scheduling discipline (SNG) (Golestani, 90) that uses packet framing to provide bounded jitter data transmission in packet-switched networks. At the same time, we developed the Hierarchical Round Robin service discipline (HRR) (Kalmanek, Kanakia, Keshav, 90) that also implements packet framing, and is rather similar in spirit (Zhang, Keshav 91). However, neither

HRR nor SNG is clearly superior to the other, as shown in Table 1. While HRR is implementable at high speeds, it cannot provide delay-jitter bounds independent of the number of hops. Thus, we feel that it would be useful to extend HRR to provide these bounds.

Feature	HRR	Stop-and-Go
High speed implementation known	Yes	No
Protects a user from other users	Yes	No
Provides delay jitter bound independent of number of hops	No	Yes

Table 1: Comparision of HRR and SNG

This was also attempted recently by Kalmanek and Morgan (Kalmanek and Morgan, 91), but while they sketched methods to synchronize skewed input frames and compensate for non-negligible switching delays, they did not describe how to implement the frame timing mechanism required by SNG. This memo shows how a simple extension of HRR can handle timing issues. Further, we present two other schemes for matching skewed frames that are easier to implement.

We first describe the operation of HRR and SNG, and then present an outline of our scheme for integrating the two disciplines when there is only one input line (or all the input lines carry synchronized frames) and only one output line (or all the output lines carry synchronized frames). This is followed by a more detailed description of the implementation, and a sketch of a proof of correctness. We then consider some details regarding frame synchronization and implementation costs.

2. HRR and Stop-and-Go

The Stop-and-Go service discipline aims to preserve the ‘smoothness’ property of traffic as it traverses the network. Time is divided into frames. In each frame time, only packets that arrived at a server in the previous frame time are sent. It can be shown that with this scheme, a user can obtain a jitter bound independent of the number of hops traversed by the virtual circuit.

A HRR server provides rate-controlled service to virtual circuits: that is, it guarantees that each circuit will receive a minimum share of the trunk bandwidth, and that no circuit will be allowed to exceed its share. A HRR server provides several levels of service, and circuits at each level are served round-robin. Conceptually, each circuit is assigned some number of *slots* in a service level, and the server cycles through the slots at each level. The time a server takes to service all the slots at a level is called the frame time at that level. The key to HRR lies in its ability to give each level a constant share of the bandwidth. Each slot at a higher level represents more bandwidth than a slot at a lower level (a user assigned to a higher level will be one that requires more bandwidth than one assigned to a lower level), so the frame time at a higher level is smaller than the frame time at a lower level.

Since a server always completes one round through its slots once every frame time, in cooperation with a leaky-bucket type of traffic shaping mechanism, it can provide an upper bound on the delay suffered by cells belonging to the channels allocated to that level. However, since a cell might arrive at a server just in time to get service, it might get a queueing delay of zero. Since the upper bound is twice the frame time, the delay jitter at each switch can be as large as twice the frame time. This can be repeated at each of a series of switches, and, in fact, on a circuit with H hops, the delay jitter can be as large as $O(H)$ (Kalmanek and Morgan, 91). This may be undesirable. In this memo, we show how a simple extension to HRR can reduce the jitter bound to twice the frame time independent of H.

3. Integrating SNG and HRR

There are two issues in integrating HRR an SNG: first, using the HRR server to provide timing information, and second, establishing a buffer pool at each output queue to allow frame synchronization. The first problem is tackled in this memo, the second was loosely described in (Kalmanek and Morgan, 91), and elaborated here.

A note on the generality of the two schemes. If a channel traverses a number of switches, then in SNG, it is assumed that the channel is assigned the same frame time at each server. If the frame time at any switch is smaller than the frame time at any previous switch, then SNG does not make any claims on the smoothness of the traffic on that channel. In contrast, HRR does not make any such assumptions. It can assign channels different frame times (levels), as well as different bandwidth shares at each successive switch, and still guarantee a delay bound as well as an upper bound on the service rate. In this work, we assume a situation that is slightly more restrictive than the one assumed by SNG, that is, that at every server along the path, a channel gets allocated the same number of slots on the same length frame.

4. Outline of the new scheme

We present the integration scheme first under the simplifying assumption that all the input frames and all the output frames are perfectly synchronized (that is, the start and end times of all the frames coincide). This assumption is removed in §7.

A HRR server has a number of levels of service. At each level, it has a service list of Virtual Circuit Identifiers (VCIs), that it cycles through, serving cells from each VCI in turn. The data for each VCI is kept in a per-VCI queue (Figure 2). In our scheme, called HRR++, we introduce (*i*) a scheduler called the eligibility scheduler (ES), and (*ii*) a set of FIFOs, two per HRR level (Figure 3). One of the FIFOs is called Current, the other is called Eligible (Figure 4).

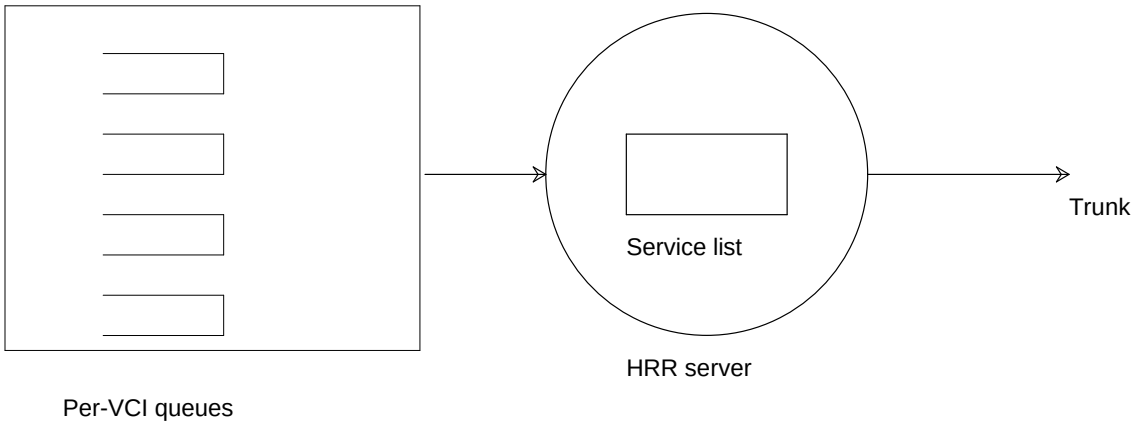

Figure 2: HRR Implementation

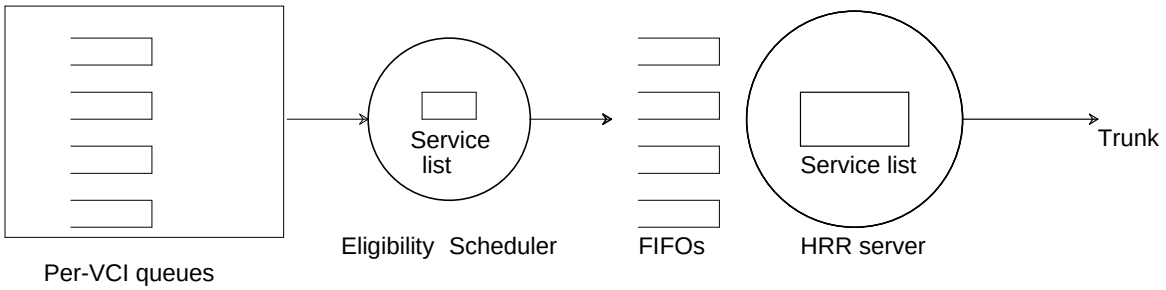

Figure 3: HRR++ Implementation

The ES and the HRR server share the service list data structure. They both use the HRR frame interleaving algorithm to decide which level to serve next. On reading a slot at that level, the ES transfers a cell from that VCI to the Eligible FIFO. Simultaneously, the HRR server transmits one cell from that level's Current FIFO. At the end of a frame, the HRR server interchanges the Current and Eligible FIFOs at that level. This is double buffering.

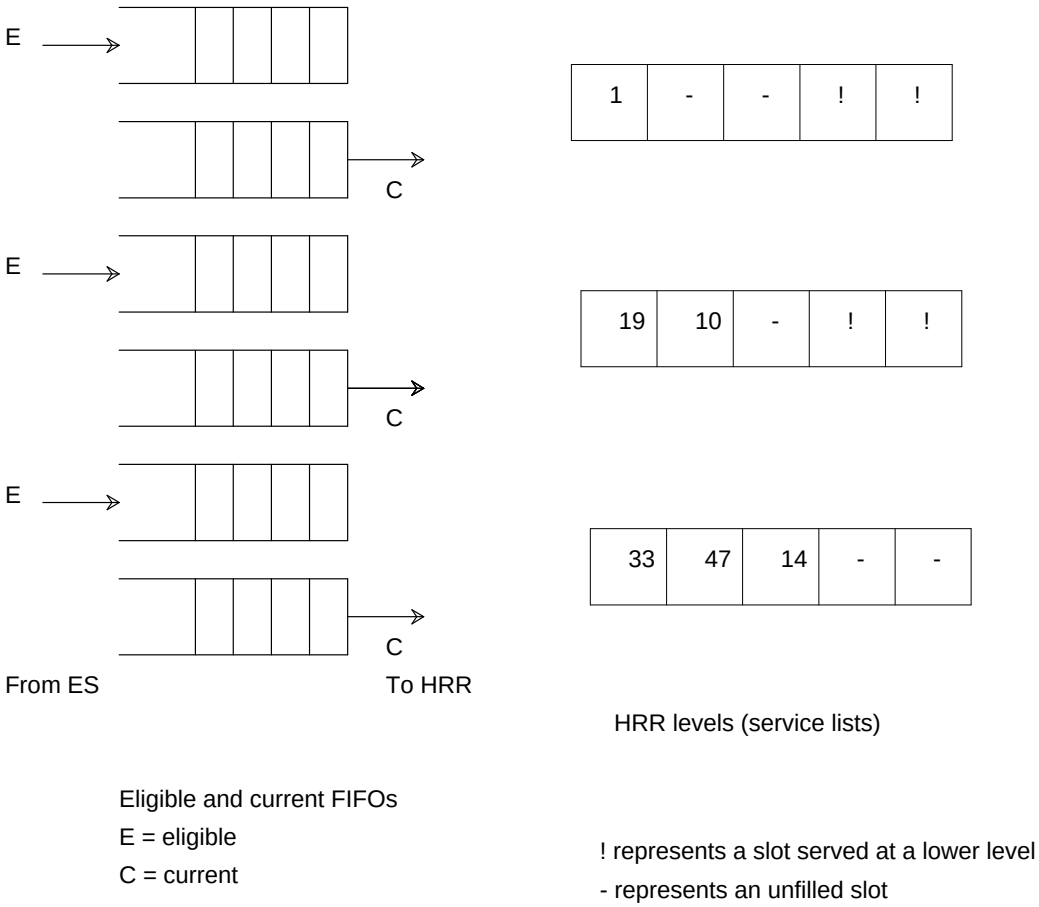

Figure 4: Eligible and current FIFOs

5. Detailed description

The components of a HRR++ server are:

1) A per-VCI queue of cells
2) The Eligibility Scheduler (ES)
3) A HRR server
4) The HRR service list data structure
5) Two FIFOs per service level

At call setup, the switch controller allocates a circuit with bandwidth, delay and delay-jitter bounds by reserving some number of slots for it at some level. This is done by writing the VCI in the service list. The admission control decision is made using an algorithm similar to the one described by (Verma, Zhang and Ferrari, 91). If the frame time at the selected level is F, and the circuit is allocated a slots in the frame, the bandwidth guaranteed is a slots per time F, the delay bound is $2F$ per hop and the delay-jitter bound is $2F$ for the whole path.

In operation, incoming data is switched through the switching fabric and placed in a per-VCI output queue (not to be confused with the FIFOs, see Figure 3). At each time step, both the HRR server and the ES read the next slot in the service list, using the frame interleaving algorithm described in (Kalmanek, Kanakia, Keshav 90). If the slot contains a valid VCI, the ES attempts to read a cell from the queue for that VCI. If it finds a cell there, it transfers one cell to the corresponding Eligible FIFO. If there are no cells, or the VCI is invalid, then no action is taken. At the same time step, one cell is read out from the Current FIFO by the HRR server, and transmitted. If the Current FIFO is empty, best-effort (non-guaranteed service) cells are served. If there are no best-effort cells, the line is left idle. The operations of the HRR server

and the ES can be done in parallel, since they are on different data paths. At the end of a frame, the HRR server switches the Eligible and Current FIFOs.

6. Proof of correctness

We show that with this scheme, a cell has both a minimum delay and a maximum delay as it traverses an HRR++ server, and that the difference between them is two frame times. Note that the service of one Current FIFO at any level always takes one frame time. Thus, on the output trunk, there is a steady stream of constant sized frames being transmitted (which are interleaved with other frame sizes as in HRR). No cell that arrives during a given frame time can leave before the start of the next frame (since it is placed in the Eligible FIFO, and not in the Current FIFO). Further, the worst delay it can get is if it just misses being written to the Eligible FIFO, in which case it may have to wait as many as 2 frame times. This is shown in Figure 5.

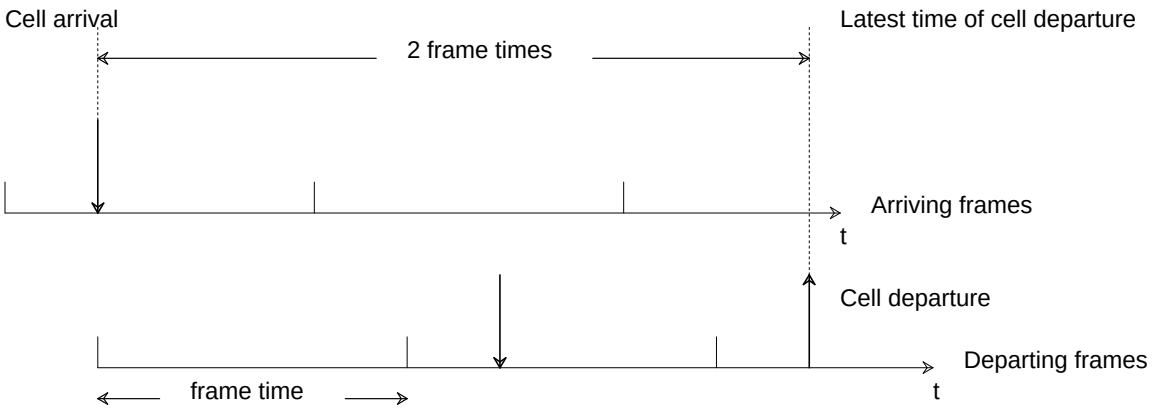

Figure 5: Worst case delay bound

In this figure, we see that a cell just missed being written into the Eligible FIFO. However, it will certainly get written into that FIFO at the latest one frame time after its arrival, and will get service at worst one more frame time later (since new VCIs join at the tail of the service list). This shows that the worst case delay bound at each switch is twice the frame time. This proves the minimum and maximum delay bounds in one switch. By an argument identical to the one used by Golestani, it is clear that over a series of switches, we get a delay-jitter bound (of twice the frame time), as well as a worst case delay bound (proportional to the number of hops in the path).

7. Frame synchronization

The scheme so far works correctly iff the input and output frames on all the trunks are perfectly synchronized. This will not be the case in practice. Thus, we have to augment our scheme to correct for unsynchronized frames on the input. Our scheme elaborates a loose sketch presented in (Kalmanek and Morgan, 91). We initially assume zero delay in the switching fabric, and later extend the scheme to cope with variable delays.

First, note that if the input and output frames are unsynchronized, we will need three FIFOs per level per output trunk. This is clear from Figure 6, which shows frames at the output trunk and the corresponding frames on input trunks. A number, such as '1', indicates that cells from that input frame must be transmitted on the similarly marked output frame. Note that at the time marked 'NOW' (corresponding to the kth frame), the ES will see cells that must be transmitted either in the $k+1$th frame time, or in the $k+2$th frame time. Thus, we need two eligible FIFOs, call them Eligible 1 and Eligible 2, and the ES must choose to place cells in one of them. Cells that belong to the $k+1$th frame should be placed in Eligible 1, cells belonging to the $k+2$th frame should be placed in Eligible 2, and at the end of each output frame, the three FIFOs should be rotated. But, how should the ES decide where to place a cell? There are several solutions.

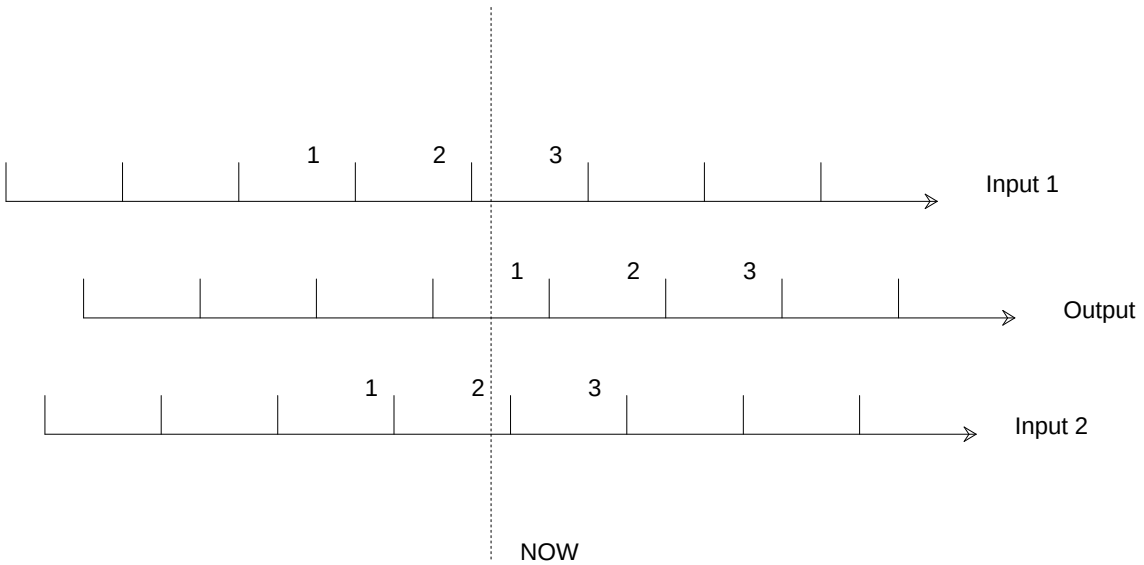

Figure 6: Need for 3 output FIFOs

7.1. Solution 1

This is the solution outlined in (Kalmanek and Morgan, 91). Each cell header is marked with a 2 bit sequence number by some entity operating at each input trunk. The sequence number cycles through the values 00, 01 and 10, with the change occurring at the beginning of each frame. The three output FIFOs at each level are correspondingly labeled 00, 01 and 10. (00 corresponds to Current, 01 to Eligible 1, and 10 to Eligible 2.) When the ES reads a VCI from the service list, it removes a cell from the per-channel queue for that VCI, and places it in the corresponding output FIFO. At the end of every output frame, the labels of the output FIFOs are rotated. It is clear that this scheme allows cells to be sorted to the correct output FIFO automatically.

The problem with the solution lies in the nature of the marking entity. In the XUNET II switch, the only possible candidate is the input Queue Module (QM). We then require the QM to determine the start of each frame at each level, and correspondingly mark headers. This can be implemented only if the QM can keep track of all the frame times simultaneously, perhaps using some kind of HRR scheme. In practice, we feel that this is not desirable, since it adds complexity to the critical path of each cell. Thus, the need for other solutions.

7.2. Solution 2

In this solution, cells are still tagged, but the marking entity is the first point in the network where framing happens, for example, the HRR server on the output trunk of the XUNET router (which is the entry point of a circuit into the network). This HRR server will be serving frames anyway, it has access to timing information, and so it is can be modified to mark the sequence number on the cell headers. Since the frame sizes are preserved through the entire network, cells from a VCI on consecutive frames will *always* have consecutive sequence numbers, no matter on which trunk they receive service. Thus, the per VCI sequence numbers can be used to identify cells from consecutive frames by the ES. The ES maintains a table called LastSeq[VCI], that maps from the VCI to the last sequence number seen for that VCI. If the sequence number seen on the cell just read differs from the LastSeq value, then the cell is put in Eligible 2, else it is placed in Eligible 1. Essentially, the change in sequence number marks the end of an input frame, and indicates that further cells should be delayed one more frame time at the output.

This solution needs a per-VCI last sequence value, since different VCIs on the same output frame may have different sequence numbers, so the direct mapping to FIFOs, as in Solution 1, is not possible. However, per-VCI queues are maintained in any case, so this information can be kept in the queue headers.

One problem with this solution is that it requires sequence bits to be carried with each cell, leading to an overhead in the bandwidth. However, since there are 8 unused bits in the congestion control field of

each ATM cell, we feel that two of them can be used for this purpose. There is another subtle problem: at the end of each output frame, the ES must update the LastSeq array. This can be computationally expensive. Alternatively, the ES must somehow determine its first visit to LastSeq in each frame, and must use the opportunity to update it. The details of the solution depend on the available hardware support, and so will not be discussed further in this memo. In any case, note that no additional complexity is introduced in the input side. Thus, we feel that this is a better solution that the earlier one.

7.3. Solution 3

The third solution does not require any tagging. Instead, the switch computes the relative synchronizations of each input and output frame, and then uses this information to process cells. This scheme is described in detail below.

We assume that each switch has its own free running clock, and that all the output frames on all the outgoing trunks are synchronized. We also assume that the switching delay between any input trunk and any output trunk is constant. In order to place a cell in the correct output FIFO, the ES needs to know the *relative phase difference* (defined below) between each input trunk and the common phase of the output trunks. The *absolute* phase difference between an input frame and output frame is the delay, in units of time, between the end of a frame in the input, and the start of the adjacent frames at the outputs. The relative phase difference between frames at the input and the output is the absolute phase difference modulo one frame time, expressed in units of one frame time. The relative phase difference always lies between 0 and 1. In general, the relative phase difference depends on the phase of the input trunk, as well as the switching delay along the path from the input to the output. For example, if the end of a frame at the input corresponds to a point that is midway through a frame at the output, and the switch fabric takes 1.2 frame times to switch a cell, then the relative phase difference is 0.7 frame times. In the discussion below, the switch delay is assumed to be 0, but the discussion holds for non-zero switching delays also, as long as the switching delay from input to output is *constant*.

In the two earlier schemes, since cells are explicitly tagged, the phase difference value is implicitly known, and does not need to be computed. In this scheme, we first compute the absolute phase difference between the superframes at the input and output trunks. This is done by marking the end of each superframe on each input trunk. When the mark reappears on the output trunk, after passing through the switching fabric, the ES knows the absolute phase difference between the input and output superframes.

The next step is to compute, for each frame size, the phase difference between the input frame and the output frame. Since the start of each input and output subframe is uniquely determined by the sizing of the HRR service lists (which, by assumption, will be common to all the servers along the path), given the superframe phase difference, the relative phase differences for each subframe can also be computed. This is the absolute phase difference of the superframes added to the start time of the first frame at that level, modulo the frame time. This value, expressed in terms of frame size, is stored in a table Phase[trunk][level] (the trunk-phase table), as shown in Figure 7. A separate table maps from each VCI to an input trunk (the VCI-trunk table). The trunk-phase table is updated at the start of each (or in general, every *n*th) superframe on the input trunk, and the VCI-trunk table is updated during call setup and teardown. We use two tables instead of one, since this allows us to update the frame phases at the start of each superframe, without having to track down every VCI on that frame.

In operation, the ES keeps track of the fraction of the superframe that has been served so far. This value, added to the start time of the first frame at that level, modulo one frame time, is the relative fraction of each frame that has been served, that is, the current phase of service for that frame. When the ES is asked to transfer a cell, it reads the VCI from the service list, and determines the input trunk from the VCI-trunk table. Then, it determines the relative phase difference from the trunk-phase table. If the relative phase difference for the VCI is ϕ, and the current phase for that frame is ξ, then

Input trunk	level	Phase
1	1	0.1
1	2	0.4
1	3	0.5
2	1	0.4
2	2	0.3

Trunk-phase table

VCI	Input trunk
123	1
213	2

VCI-trunk table

Figure 7: Synchronization tables

if $\xi > 1 - \phi$, it places a cell in Eligible 2
else, it places it in Eligible 1.

It is easily shown that this has exactly the same effect as marking each input cell.

The cost of this solution lies in computing the relative phases of the input and output frames, and in maintaining the current phase information. The current phase for each frame size can be determined by the cell count since the start of the superframe, the start time of the first frame at that level, and the size of a frame. If n cells have been served thus far, the first frame at that level is served after l cells, and the frame size at some level is f cells, then the phase is simply $(n + l) \bmod f$. The relative phase difference between the input and output frames can be similarly computed.

7.4. Non-zero switching delays

Thus far, we have assumed that the switching delay is zero. If the switching delay is non-zero but constant, then the frames arriving at the output trunk will be delayed by a constant factor, and this is exactly as if the preceding trunk had an additional propagation delay equal to the switching delay. Hence, the three solutions described above will work with no additional modifications.

What if the switching delay is variable? Following the model of (Kalmanek and Morgan, 91), we assume that the switching delay between any input trunk and any output trunk is bounded from above by a value Δ. Also, we define p to be an integer such that at each level Δ lies between $p-1$ and p frame times (so p is different for each level). With this model in hand, we re-examine the three solutions.

It is clear that if the delay is variable, at any moment, cells may arrive that may belong to as many as $p + 3$ frames. Hence, for all three solutions, we will need to maintain $p + 3$ FIFOs per output level. The solutions differ only in the algorithm used to determine in which FIFO to place a cell.

For the first solution, the obvious extension is to increase the tag field from 2 to ceil(ln p) bits. The sequence numbers cycle from 0 to $p + 2$. The output trunk simply reads off the tag value, and places the cell in the FIFO that corresponds to the tag value. At the end of the frame, the labels of the FIFOs are rotated. As before, the complexity lies in the input trunk, which will have to tag the cells somehow.

The second solution can be similarly extended. The additional cost is that the LastSeq[VCI] array will have to hold more bits. Also, the bandwidth overhead will be larger.

The third solution will not work if the switching delays can be variable, since the ES has no way to differentiate between cells destined for different FIFOs. However, if the switching delay is relatively small, then the phase computation can be done pretending that the switching delay is zero, and the resulting error will be small. If the delays are non-negligible, the ES could send cells to Eligible 1 till half of the output

frame, and to Eligible 2 thereafter. Assuming that all possible frame synchronizations are equally likely, cells on any virtual circuit will be served too early or too late with equal likelihood, so that expected delay jitter will converge as the length of the path goes to infinity. In a network with a large number of hops, this may be a reasonable approach.

8. Implementation cost

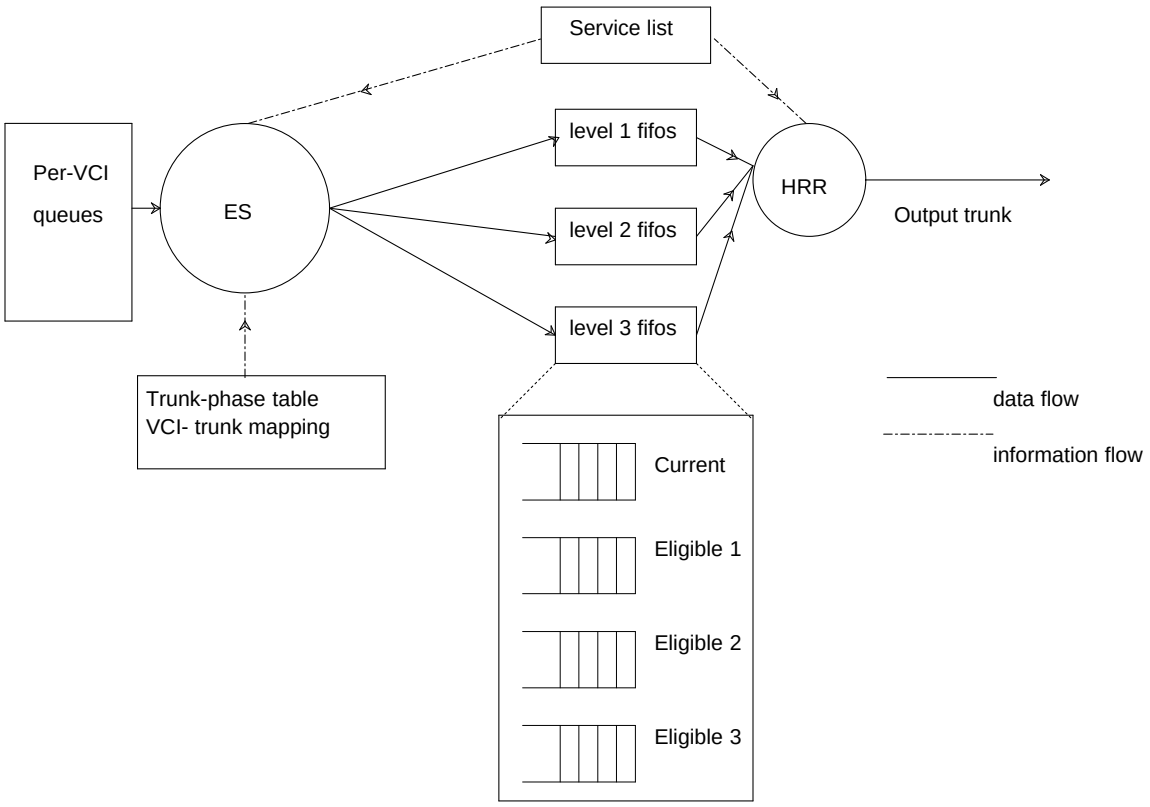

Figure 8: Complete scheme

The complete scheme is presented in Figure 8. The cost of the scheme is basically in
1) Providing more FIFOs
2) Implementing the ES
3) Synchronization control

If the switching delays are constant, we will need 3 FIFOs per level, and the depth of a FIFO is the number of slots at each level. This will probably be not very large (on the order of a few kilobytes). Thus the memory cost is small. Even if the number of FIFOs is increased to p+3 per level, as long as p is reasonably small (we expect it to be 1 or 2, for most switches), the cost is not appreciably more.

The ES is implemented as a simple extension of the HRR server. It only involves one more operation at each step of the HRR server, and this can be done in parallel with the other operations, since they do not share a data path.

The cost of synchronization depends on the solution used. For the first solution, the cost is the complexity in the input queue module. This may be prohibitively high. For the second solution, the cost is the overhead for sequence information, the complexity of the marking scheme at the router, and the per-VCI sequence information. For a small enough p, the loss of bandwidth is on the order of a few bits per cell, and this seems reasonable. Since per-VCI information can be kept with each VCI queue header, this cost also is small. The marking scheme at the router needs the HRR server to be augmented to mark outgoing frames with a 2 or 3 bit header. This can be done relatively easily. Thus, the overall cost of the scheme does not appear to be too much.

The third scheme needs computation of relative phases, and keeping track of the current phase. Since the first step is done infrequently, it can be performed offline by the switch controller. Keeping track of the current phase can be done simply by counting the number of cells transmitted since the start of the current superframe, and this is easy. Thus, the third scheme is the most attractive one. However, since it does not work correctly in networks where switches can have variable switching delays, in such networks, we prefer the second scheme.

9. Limitations

We assume that the service hierarchy at all the servers, and the trunk speeds of all the trunks, are the same. If this does not hold, then the scheme fails. Extending the scheme to allow for such variations (which would probably occur in practice) is a topic for future work.

Another assumption is that the clocks on the input and output trunks do not drift appreciably, so that frame times remain relatively constant. If the clock frequencies can drift with respect to each other, then the frame times at the input and output trunks will not match, and this introduces further delay-jitter. Since clocks in DS3 trunks drift only about one cell every 5 minutes, if the drift is corrected at the physical or datalink layer, this should not pose a severe problem to stop-and-go service.

10. Conclusions

We have presented detailed schemes to implement Stop-and-Go queuing in networks of HRR servers, such as XUNET II. We have shown that HRR servers can provide the timing information that is crucial to implement SNG. Further, we have examined requirements for frame synchronization in detail. We have discussed three alternate solutions for the problem, which are appropriate for switches with constant, or variable but bounded, switching delay. In light of our work , we feel that implementing a HRR++ server is practical, and probably not much more complex than implementing a HRR server.

11. Acknowledgments

This work owes much to discussions with S.P. Morgan and C.R. Kalmanek.

12. References

(Golestani, 90): S.J. Golestani, "A Stop-and-Go Queueing Framework for Congestion Management", Proc. ACM SigComm 1990, September 1990, pp 8-18.

(Kalmanek, 91): C.R. Kalmanek, Personal communication, September 1991.

(Kalmanek, Kanakia, Keshav, 90): C.R. Kalmanek, H. Kanakia and S. Keshav, "Rate-Controlled Servers for Very High Speed Networks", Proc. Globecom ’90, December 1990.

(Kalmanek and Morgan 91): C.R. Kalmanek and S.P. Morgan, "Combining Stop-and-Go Queueing with Hierarchical Round Robin Service in an ATM Network", Bell Labs Internal Memorandum 11275-910426-03TM, April 1991.

(Verma, Zhang and Ferrari, 91): D. Verma, H. Zhang and D. Ferrari, "Delay Jitter Control for Real-Time Communication in Packet Switching Networks", Proc. Tricomm’91 Conf., April 1991.

(Zhang, Keshav, 90): H. Zhang and S. Keshav, "Comparision of Rate-based Service Disciplines", Proc. ACM SigComm 1991, September 1991.