

# Packet Stream SPS Architecture: Scheduler

---

Victor Ramamoorthy

Draft version

March, 2000

## Contents

<b>PACKET STREAM SPS ARCHITECTURE: SCHEDULER .....</b>	<b>1</b>
CONTENTS .....	2
INTRODUCTION.....	3
SPS ARCHITECTURE .....	3
SCHEDULING BASICS.....	4
QUANTIZED LEHOCZKY-SHA-DING TEST FOR SCHEDULABILITY .....	5
<i>Normalization</i> .....	5
<i>Test for schedulability:</i> .....	5
<i>Code Fragment for schedulability test</i> .....	6
<i>Schedule Cycle Length</i> .....	7
<i>Schedule Table Length Computation</i> .....	7
<i>First 170 Prime Numbers</i> .....	9
<i>Period Transformation</i> .....	11
<i>Schedule Cycle Generation</i> .....	11
PUTTING IT ALL TOGETHER.....	12
WHAT WE HAVE ACCOMPLISHED SO FAR .....	13

## Introduction

### SPS Architecture

The aim of the SPS architecture is to provide a flexible delivery of packets with guaranteed parameters of QOS. The ideas are best explained by looking at figure 1 where the system block diagram is shown.

The entire system can be decomposed as four different subsystems. The *input port* receives packets from a network and sends them over to *front-end packet preprocessor*. The front-end preprocessor does a number of different preprocessing operations: First it classifies the incoming packet according to a well-defined *classification criteria*. The result of the classification can lead to a formation or modification of a *flow* of packets in a *virtual circuit* or a *virtual path*. Secondly, the preprocessor can measure the input rate and make appropriate decisions. This *metering* function helps in managing the internal queues and to detect unruly traffic conditions. The preprocessor also performs traffic policing to allow only well behaved packet streams. The preprocessor can perform several *admission control* strategies. Finally the preprocessor can also accomplish *shaping the traffic* to fit desired properties. This document does not explain the details of the front-end preprocessor design.

What comes out of the preprocessor are the following: (1) a set of packet streams requiring quality of service. The quality of service is specified in terms of desired bit rate and desired delay (2) a set of packet streams with no associated QOS guarantees. The streams with QOS need careful attention and work. The streams without QOS are treated with the *best effort* approach. The front-end preprocessor also sends and receives messages from host processor. The host runs software that has intelligence to control the preprocessor.

The third block in the architecture is the most critical part: It is the *QOS Scheduler, Smoother and Delivery Engine*. This documents deals with the design of this block. Without this, the whole system collapses. As the name implies, this block schedules packet stream delivery to the fourth subsystem, namely the *output port*.

The job of the preprocessor is to create packet streams with and without QOS to the scheduler block. A packet stream can be just a virtual circuit or a virtual circuit in a virtual path or a combination of many virtual circuits and virtual paths. A packet stream issued from the preprocessor can have a QOS criterion or can be a "normal" best effort variety. The scheduler does not care if an input stream comes from one virtual circuit or from many. It just looks at the set of QOS streams, each specified with a bit rate and delay, attempts to create a schedule that will satisfy the requirements on the whole. While creating the schedule, this block also gives an indication of *residual bandwidth* (i.e., the output port bandwidth left after scheduling all the QOS streams) and attempts to perform "best effort" delivery of the packet streams without QOS.

It is quite possible that the QOS requirement cannot be met for a given set of streams with QOS. In such situations, the scheduler block conveys this to the host software, which can make a higher-level decision. If the QOS requirement can indeed be met, then the scheduler generates a *schedule table* that governs how packets are to be shipped to the output port. The host software can also modify or fine-tune the schedule table to suit different implementation requirements. It can also command the scheduler to perform iterative searches until a given requirement is met.

The packet delivery satisfying the QOS criteria (of the QOS streams) is simply done by sending packets of an input stream to the output port for a well-defined interval called *time slice*. During the course of a time slice, only one stream is sending the packets to the output port. The schedule table decides connection to the output port and picks up packets from an appropriate stream for the duration of a time slice. The entries in the schedule table just tell which stream is sent to the output port during which time slice. The scheme shown in fig.2 gives the details. The generation of the schedule table can be done at any time: in one

strategy, it can be done when changes in input packet streams is detected; in another, it can be done periodically ,for example, every 50 milliseconds.

## Scheduling Basics

Let us now formalize the problem as follows: We have a set of packet streams, collectively called  $\alpha_n$ , consisting of  $n$  packet streams  $\{a_1, a_2, a_3, \dots, a_n\}$  with each stream  $a_i$  is described by its QOS parameter set  $\tau_i$  which is described by the doublet  $\{R_i, d_i\}$  where  $R_i$  is the average bit rate (bits/sec) to be maintained for the stream  $a_i$  with a maximum delay  $d_i$  seconds. Let the output port operate with a bit rate of  $R_{out}$  bits/sec, which is greater than any stream average bit rate  $R_i$ . That is, we require that  $R_{out} \gg R_i, \forall i$ . For each stream  $a_i$ , it is then natural to assign a period  $T_i \leq d_i$  so that the stream is serviced in a periodic manner to guarantee the requirement of average bit rate  $R_i$ . To achieve this average bit rate of  $R_i$  for the stream  $a_i$ , the stream must be connected to the output port for a duration of  $C_i$  seconds such that  $R_i = \frac{C_i R_{out}}{T_i}$ . Since the output port is connected to the stream  $a_i$  only for  $C_i$  seconds in every  $T_i$  second, the stream  $a_i$  needs to buffer only  $C_i R_{out}$  bits in its queue  $Q_i$ .

Since our requirements of maintaining the average bit rate can be met as long as the output port gets connected for  $C_i$  seconds, it is not necessary it stays in the same place (same phase) for every period interval of  $T_i$  seconds. All the servicing tasks in the stream set  $\alpha_n$  are periodic and have a *deadline* requirement of finishing the packet transmission *within* their respective task periods,  $\{T_i\}$ . If all the tasks in  $\alpha_n$  meet their deadlines, then  $\alpha_n$  is said to be *schedulable*. What we have done so far is to reduce the original requirement of meeting the QOS parameters  $\{R_i, d_i\}$  for the entire set  $\alpha_n$  to an equivalent parameter set called *task set*  $\{C_i, T_i\}$ .

Let  $U_n$  be the utilization of the output port for a stream set  $\alpha_n$  is given by  $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$ . Because the output port can not send more than  $R_{out}$  bits/sec at any time, this quantity (as a sum of ratios) must be less than or equal to unity. The periods  $T_i$  and computation  $C_i$  are given in milliseconds or seconds. If  $\alpha_n$  is schedulable, then a derived set from  $\alpha_n$  consisting of only  $n-k$  streams obtained by removing  $k$  streams of  $\alpha_n$ , called  $\alpha_{n-k}$  is also schedulable for all  $0 \leq k \leq n$ . This also means that  $U_{n-k} \leq U_n$ .

If there are two schedulable stream sets  $\alpha_n$  and  $\beta_m$ , and if a third stream set  $\delta_k = \alpha_n \cap \beta_m$  is constructed by selecting applications from both  $\alpha_n$  and  $\beta_m$ , then  $\delta_k$  may or may not be schedulable. Note that there are  $2^{n+m}$  possible ways of creating the new stream set  $\delta_k$  and each combination needs separate testing for schedulability.

Note that the condition  $\sum_{i=1}^n \frac{C_i}{T_i} < 1$  may appear to be sufficient to imply that a stream set is schedulable<sup>1</sup>.

However this is not true. An stream set may satisfy the condition  $U_n = \sum_{i=1}^n \frac{C_i}{T_i} < 1$  and yet may not be schedulable. Liu and Layland<sup>2</sup> derived a least upper bound on utilization equal to  $n(2^{1/n} - 1)$  for a stream set  $\Gamma_n$  with  $n$  periodic tasks. For a large number of tasks, this bound approaches the value of 0.69. The implication is that only task sets with 69% of processor utilization are schedulable. Any task set with total utilization less than this bound could be scheduled by the Rate Monotonic algorithm defined below. However some task sets that are *above* this bound may also happen to be schedulable.

### **Quantized Lehoczky-Sha-Ding Test for Schedulability**

The real crux of scheduling problem is finding when a task set is schedulable. It took more than decade to answer the above question since the original formulation by Liu and Layland. Here we present a discrete version of the Lehoczky-Sha-Ding<sup>3</sup> test that is of importance in the present context. Though the periods and connection intervals for a given stream are continuous variables, it is necessary to quantize them with a time interval called Time-Slice Duration,  $T_{Time-Slice}$ . This is because, during the time-slice interval, a stream is connected to the output port and packets are pulled out from its queue and transmitted out. At the end of this interval, a decision to switch to another stream or not is made.

#### Normalization

Given a task set  $\Gamma_n = \{\tau_1, \tau_2, \dots, \tau_n\}$ , find a normalized task set  $\hat{\Gamma}_n = \{\hat{\tau}_1, \hat{\tau}_2, \dots, \hat{\tau}_n\}$  where

$$\hat{\tau}_i = \left\{ \mathcal{Q}[C_i] = \left\lceil \frac{C_i}{T_{Time-Slice}} \right\rceil, \mathcal{Q}[T_i] = \left\lceil \frac{T_i}{T_{Time-Slice}} \right\rceil \right\}, \text{ where the notation } \lceil x \rceil \text{ means the greatest}$$

integer closest to  $x$ .<sup>4</sup> The algorithm for testing the schedulability is then an iterative one:

#### Test for schedulability:

1. Compute  $t(0) = \sum_{i=1}^n \mathcal{Q}[C_i]$

<sup>1</sup> In fact, considerable effort was focused on relating scheduling and utilization. Strictly speaking, utilization bound depends on the task model. For multiframe task models, the Liu-Layland utilization can be more than unity and still the task set could be schedulable.

<sup>2</sup> C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment", *Journal of the ACM*, Vol.20, No.1, January 1973, pp. 40-61.

<sup>3</sup> John Lehoczky, Lui Sha and Ye Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", *Proceedings of the IEEE Real-Time Systems Symposium 1989*, IEEE Computer Society Press, pp 201-209.

<sup>4</sup> That is, if  $x$  consists of an integer part  $\lceil x \rceil$  and a fractional part  $(x)$ , then  $\lceil x \rceil = \begin{cases} \lceil x \rceil & \text{if } (x) = 0 \\ \lceil x \rceil + 1 & \text{if } (x) \neq 0 \end{cases}$

$$2. \text{ Compute } t(i) = \sum_{i=1}^n Q[C_i] \cdot \left\lceil \frac{t(i-1)}{Q[T_i]} \right\rceil \text{ for } i = 1, 2, 3, \dots$$

3. if  $t(i+1) = t(i)$  and  $t(i+1) \leq Q[T_n]$ , then the task  $\Gamma_n = \{\tau_1, \tau_2, \dots, \tau_n\}$  is schedulable.

The convergence in step 3 above is necessary to guarantee schedulability. The main idea of the testing is to release all the tasks simultaneously (that is, *the critical instant*) and check if the task with the largest period still meets its deadline equal to its period. At the critical instant, the task with the largest period has the longest waiting time before it starts executing. If the task with lowest priority can meet its deadline, then all the other tasks with higher priorities are bound to meet their deadlines, and hence the task set is schedulable. Note that the testing is fast even for a large number of tasks. This test can also be implemented either in hardware or as a part of host control software.

## Code Fragment for schedulability test

Here is a “C-like” code fragment that does the above test:

```
double time, test, ratio;
int l,j;
int ivar,factor;
boolean lsdresult;

int QC[1 to 100], QT[ 1 to 100]; // contain quantized periods and connection times
int N; // actual number of streams

if (N!=0)
{
    time = 0.0;
    for(l=1; l <= N; l++)
    {
        time += QC[l];
    }

    looplsd: test = 0;
    for(l=1; l <= N; l++)
    {
        ratio = time/QT[l];
        ivar = (int)(time/QT[l]);
        if((ratio - ivar) > 0)
        {
            factor = ivar + 1;
        }
        else
        {
            factor = ivar;
        }
        test += QC[l];
    }
    if(test != time && test <= QT[N])
    {
        time = test;
        goto looplsd;
    }
    else
    {
        if(test <= QT[N] && test ==time)
        {
            lsdresult = true; // Test passed
        }
    }
}
```

```

    }
    else
    {
        Isdresult = false; //Test failed
    }
}
else
{
    Isdresult = false;
}

```

## Schedule Cycle Length

Since the normalized task set has integers as periods, it is possible to store the schedule - which is also cyclical - as a *Schedule Cycle Table* or simple called as *Schedule Table*. The length of the schedule table is of importance, because it needs storage space in SRAM<sup>5</sup>. The schedule cycle length is given by the least common multiple of  $\{Q[T_1], Q[T_2], \dots, Q[T_n]\}$ . This is so because tasks are all periodic. They tend to have harmonic consonance and dissonance intervals. The smallest interval separating two consonance of all tasks is the schedule cycle length. At the consonance, all the tasks are appear simultaneously, that is creating a critical instant. If the tasks all meet their deadlines at consonance, they are bound to meet their deadlines at all other times. That is why the critical instant plays an important role in the scheduling theory. Schedule Cycle length can be found in the following manner:

- Express  $Q[T_i] = p_1^{e_1(i)} \cdot p_2^{e_2(i)} \cdot p_3^{e_3(i)} \dots p_M^{e_M(i)}$  for  $i = 1, 2, 3, \dots, n$ , where

$p_1, p_2, p_3, \dots, p_M$  are the first  $M$  prime numbers<sup>6</sup>. From the numerical range point of view,  $M$  is chosen to be 170. (This means that we may need to store first 170 prime numbers either in the host software or in the SRAM.) Any integer can be expressed in the above manner by Prime Number factorization theorem. The exponents  $e_1(k), e_2(k), e_3(k) \dots e_M(k)$  can be zero or positive numbers.

- The Schedule Cycle length is given by
 
$$L = p_1^{\max_{i=1,2,3,\dots,n}\{e_1(i)\}} \cdot p_2^{\max_{i=1,2,3,\dots,n}\{e_2(i)\}} \cdot p_3^{\max_{i=1,2,3,\dots,n}\{e_3(i)\}} \dots p_M^{\max_{i=1,2,3,\dots,n}\{e_M(i)\}}$$

The prime number factorization required in step 1 can be done by a modified Eratosthenes Sieve algorithm<sup>7</sup> and an implementation<sup>8</sup> is given below:

## Schedule Table Length Computation

<sup>5</sup> It is conceivable that the schedule cycle table can be constructed on the fly without any storage. But computing the table first helps in reducing the complexity of implementation. Another possibility is that the host software can compute the schedule table partially and store it SRAM.

<sup>6</sup> This can be also thought of as taking a prime-spectrum of any integer.

<sup>7</sup> <http://www.utm.edu/research/primes/index.html#primes>

<sup>8</sup> One simple implementation simply tests for all prime numbers. We need to use the prime number sieve only for square root of the input number for primality.

The following “C-like” code fragment illustrates a simple way to compute the schedule table if stream set is found to be schedulable:

```

Int prime[1 to 170],expo[1 to 170],work[1 to 170]; // prime array contains the prime table given below
                                                    // other arrays carry temporary variables
int inputvar, integervar, remainder,seqlength;
int QT[1 to 100], QC[1 to 100]; // quantized periods and connection times which schedule test

// Initialize
for(l=1; l <= 170; l++)
{
    work[l] = expo[l] = 0;
}

for(l = 1; l <= N; l++)
{
    inputvar = QT[l]; // Save it for future
                    // check if it is prime
    for(j=1; j<= 170; j++)
    {
        if (inputvar == prime[j])
        {
            expo[j] = expo[j] + 1;
            goto yesprime;
        }
    }

    Notprime: for(j=1; j <=170; j++)
    {
        work[j] = 0;      // Reset working array
    }

    breakup: for(j=1; j <= 170; j++)
    {
        integervar = inputvar/ prime[j];
        remainder = inputvar % prime[j]; // we take mod here

        if(remainder ==0 && integervar > 1)
        {
            work[j] = work[j] + 1;
            inputvar = integervar;
            goto breakup;
        }

        if(remainder == 0 && integervar ==1)
        {
            work[j] = work[j] + 1;
            goto finished;
        }
    }

    finished: for(j=1; j <= 170; j++)
    {
        if(work[j] > expo[j]) expo[j] = work[j];
    }

    yesprime: Continue;
}

seqlength = 1;
for(l=1; l <= 170; l++)
{
    if(expo[l] > 0)

```

```
    {
        for(j=1; j <= expo[l]; j++)
        {
            seqlength = seqlength*prime[l];
        }
    }
}
// seqlength contains the schedule table length
```

## First 170 Prime Numbers

Here is the list of first 170 prime numbers needed for the above code fragment. The array prime contains these numbers:

2	353	811
3	359	821
5	367	823
7	373	827
11	379	829
13	383	839
17	389	853
19	397	857
23	401	859
29	409	863
31	419	877
37	421	881
41	431	883
43	433	887
47	439	907
53	443	911
59	449	919
61	457	929
67	461	937
71	463	941
73	467	947
79	479	953
83	487	967
89	491	971
97	499	977
101	503	983
103	509	991
107	521	997
109	523	1009
113	541	1013
127	547	
131	557	
137	563	
139	569	
149	571	
151	577	
157	587	
163	593	
167	599	
173	601	
179	607	
181	613	
191	617	
193	619	
197	631	
199	641	
211	643	
223	647	
227	653	
229	659	
233	661	
239	673	
241	677	
251	683	
257	691	
263	701	
269	709	
271	719	
277	727	
281	733	
283	739	
293	743	
307	751	
311	757	
313	761	
317	769	
331	773	
337	787	
347	797	
349	809	

## Period Transformation

Though the above model fits well into multimedia applications, there may exist need for changing the priority of different tasks independent of the periods. A simple period transformation is useful in this context. Given a task  $\tau_i = \{C_i, T_i\}$ , its priority is determined by the period  $T_i$ . Now by applying a period transformation of  $\hat{\tau}_i = \{a \cdot C_i, a \cdot T_i\}$ , where  $a$  is a non-negative scale factor, the task priority can be changed within the task set  $\tau_i$ . This facility is useful in finding a suitable design.

## Schedule Cycle Generation

Rate Monotonic Algorithm simplifies the schedule cycle generation to a great extent. In fact the following "C-like" code fragment can be used for Schedule Cycle generation:

```
int schedule[ 1 to L];           // schedule is an array which holds the final schedule;
                                // L is the schedule Cycle Length;
int QT[1 to n], QC[1 to n];     // There are n tasks with periods QT[i] and connection time QC[i]
                                // Note: These arrays are sorted according to ascending order of QT[i].

int noofcycles;                 //holds number of cycles of a task within a schedule cycle
int address, addressmax,addressmin; //address pointers
int i,j,k,extra;

for(i=1; i <=N; i++)
{
    schedule[i] = 0;           // Initialize the array to zero
                                // Any entry of zero means that the time slice is free and not used by
                                //QOS streams. This time slice can be used to squeeze in other non
                                //QOS streams. The number of zeros in this array indicates residual
                                // bandwidth.
}

for(i =1; i <=N; i++)
{
    noofcycles = L/QT[i];

    if( i==1)
    {
        for(j=1; j <=noofcycles-1; j++)
        {
            for(k=1; k <=QC[i]; k++)
            {
                address = (j-1)*QT[i] + k; // This is the highest priority task; so let it
                // have its schedule.
                schedule[address] = i;
            }
        }
    }
    else
    {
        for(j=1; j <=noofcycles-1; j++)           // This is a low priority task
                                                    // search if there is any slot open
        {
            extra = 0;                             // counter for used up time-slices
            addressmin = (j-1)*QT[i] + 1;
            addressmax = j*QT[i];
        }
    }
}
```

```

for(address=addressmin; address <=addressmax; address++)
{
    if( schedule[address] = 0 && extra < QC[i]) // look for a free time-slice
    {
        schedule[address] = i;
        extra++;
    }
}
}
}
}

```

The array schedule stores zero to indicate a free slot and task number to indicate occupancy. Hence schedule cycle generation can be done either on the fly or off-line.

### Putting it all together

Let there be  $n$  streams each with QOS requirement of  $\{R_i, d_i\}$ . Convert the requirements to equivalent task sets of  $\{C_i, T_i\}$ . From the hardware/software specifications, choose a list  $T_{List}$  of time slice intervals that are feasible. The resolution of the time slice entries be  $\delta$  milliseconds. Similarly choose a maximum schedule table size  $L_{Max}$ . For example, if  $L_{Max} = 32767$ , then 16 bit numbers will be used as address to find an entry in the schedule table.

1. Choose a time-slice duration,  $T_{Time-Slice}$  in milliseconds from  $T_{List}$ .
2. Compute  $Q[C_i] = \left\lceil \frac{\hat{C}_i^{Max}}{T_{Time-Slice}} \right\rceil$  for  $i = 1, 2, 3 \dots n$ .
3. Compute  $Q[T_i] = \left\lceil \frac{T_i}{T_{Time-Slice}} \right\rceil$  for  $i = 1, 2, 3, \dots n$ .
4. Compute  $t(0) = \sum_{i=1}^n Q[C_i]$
5. Update  $t(i) = \sum_{i=1}^n Q[C_i] \cdot \left\lceil \frac{t(i-1)}{Q[T_i]} \right\rceil$  for  $i = 1, 2, 3 \dots$
6. if  $t(i+1) = t(i)$  and  $t(i+1) \leq Q[T_n]$ ,  
then the task set is schedulable. Go to Step 8.  
Otherwise go to step 7.
7. Find new solutions by
  - (a) Use Period Transformation to scale the periods, i.e., find a scalar value  $a$ , such that

$$Q[C_i] = \left\lceil a \cdot \frac{\hat{C}_i^{Max}}{T_{Time-Slice}} \right\rceil \text{ for } i = 1, 2, 3 \dots n, \text{ and}$$

$$Q[T_i] = \left\lceil a \cdot \frac{T_i}{T_{Time-Slice}} \right\rceil \text{ for } i = 1, 2, 3 \dots n. \text{ Then go back to step 4. Or}$$

(b) Use a new value for  $T_{Time-Slice}$  and go to step 2.

8. Express  $Q[T_i] = p_1^{e_1(i)} \cdot p_2^{e_2(i)} \cdot p_3^{e_3(i)} \cdots p_M^{e_M(i)}$  for  $i = 1, 2, 3, \dots, n$ , where

$p_1, p_2, p_3, \dots, p_M$  are the first  $M$  prime numbers. Find  $L$ :

$$L = p_1^{\max_{i=1,2,3,\dots,n}\{e_1(i)\}} \cdot p_2^{\max_{i=1,2,3,\dots,n}\{e_2(i)\}} \cdot p_3^{\max_{i=1,2,3,\dots,n}\{e_3(i)\}} \cdots p_M^{\max_{i=1,2,3,\dots,n}\{e_M(i)\}}.$$

9. If  $L \leq L_{Max}$ , then go to step 10. Otherwise go to step 7.

10. Update the time-slice by picking another entry from  $T_{List}$ . Set

$$T_{Time-Slice-Old} = T_{Time-Slice}. \text{ Then Update}$$

$$T_{Time-Slice} = T_{Time-Slice-Old} + \delta.$$

If  $|T_{Time-Slice} - T_{Time-Slice-Old}| < 0.01$  or the number of times the iteration was performed was large, then stop: We have found the solution. Otherwise go to step 1.

### **What we have accomplished so far**

Our journey so far has been to find out how to get packets out satisfying QOS. The requirements have been to ensure that (1) we satisfy bit rate requirements of many streams, (2) we satisfy delay requirements of many input streams and (3) we provide smooth “channelized” flow of packets from the output port. The third requirement is automatically satisfied because of very construction. We not only provide smooth streams from the output port – we *guarantee* that the flows adhere to the specification by imposing hard real-time constraints. Whenever the input requirements can not be met with, we flag it. We let only streams that are possible to accommodate without violating physical limitation of devices. Since scheduling is independent of clock speed of hardware, we can adopt “divide and conquer” strategy to scale up with more streams and additional copy of the same hardware. Hence a solid foundation is laid out for packet delivery mechanism that will be the backbone of SPS. Now let us turn our attention to other interesting possibilities.

How to schedule ABR and UBR traffic

How to Schedule VBR traffic

How to schedule CBR traffic

How to add/drop/modify streams

Cacheing Search Solutions

Complete System



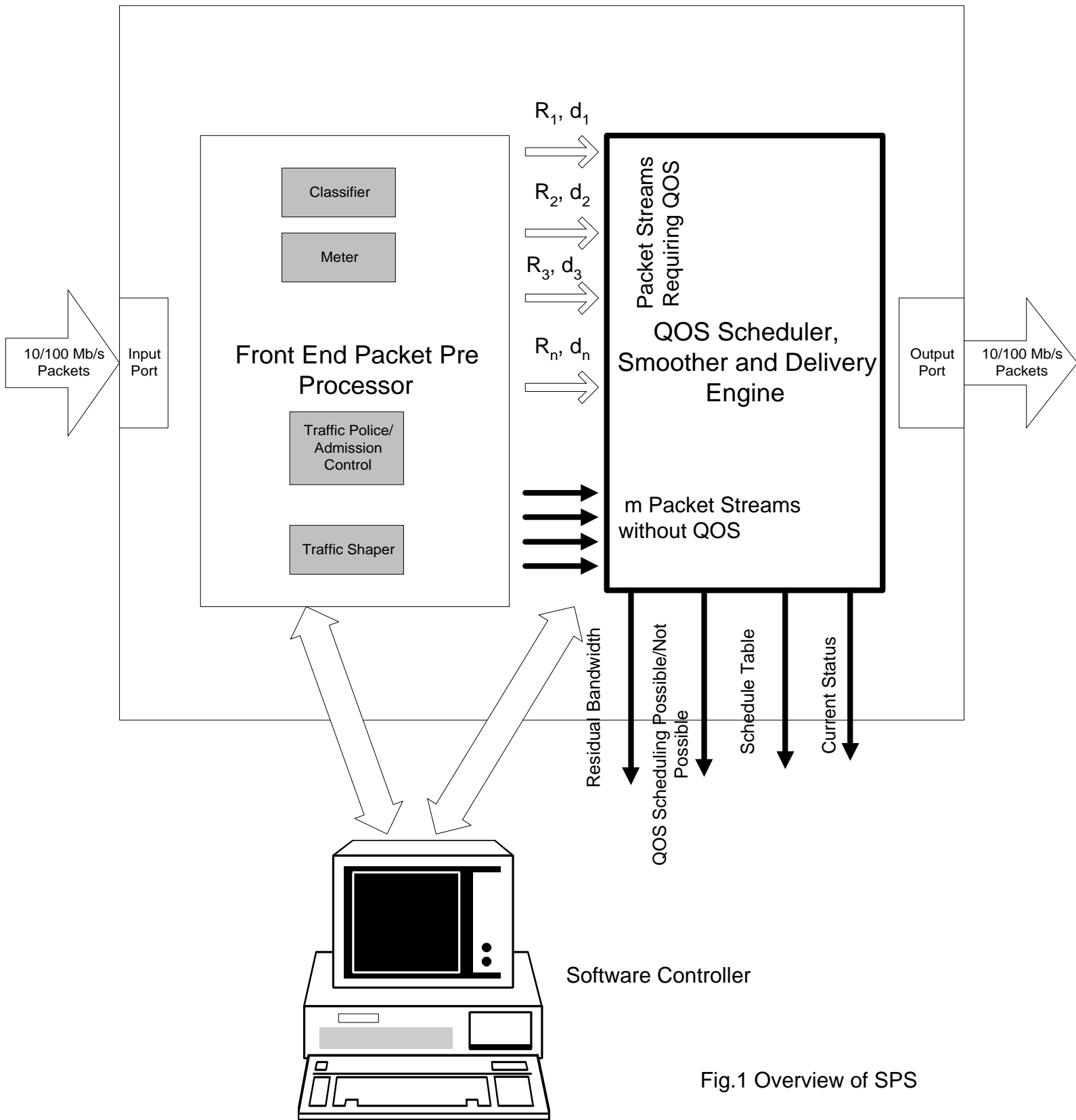


Fig.1 Overview of SPS

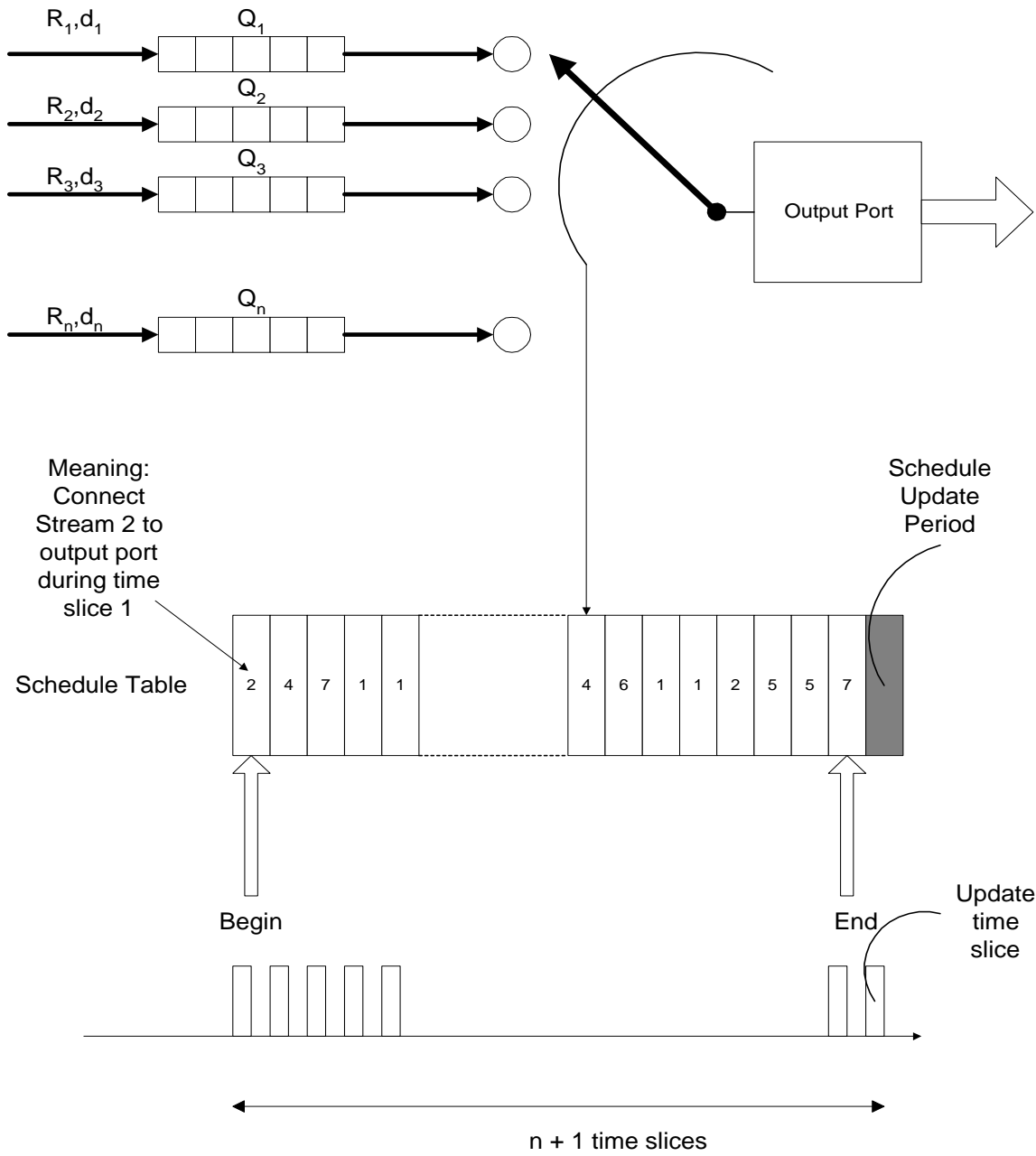


Fig.2 Schedule Mechanism

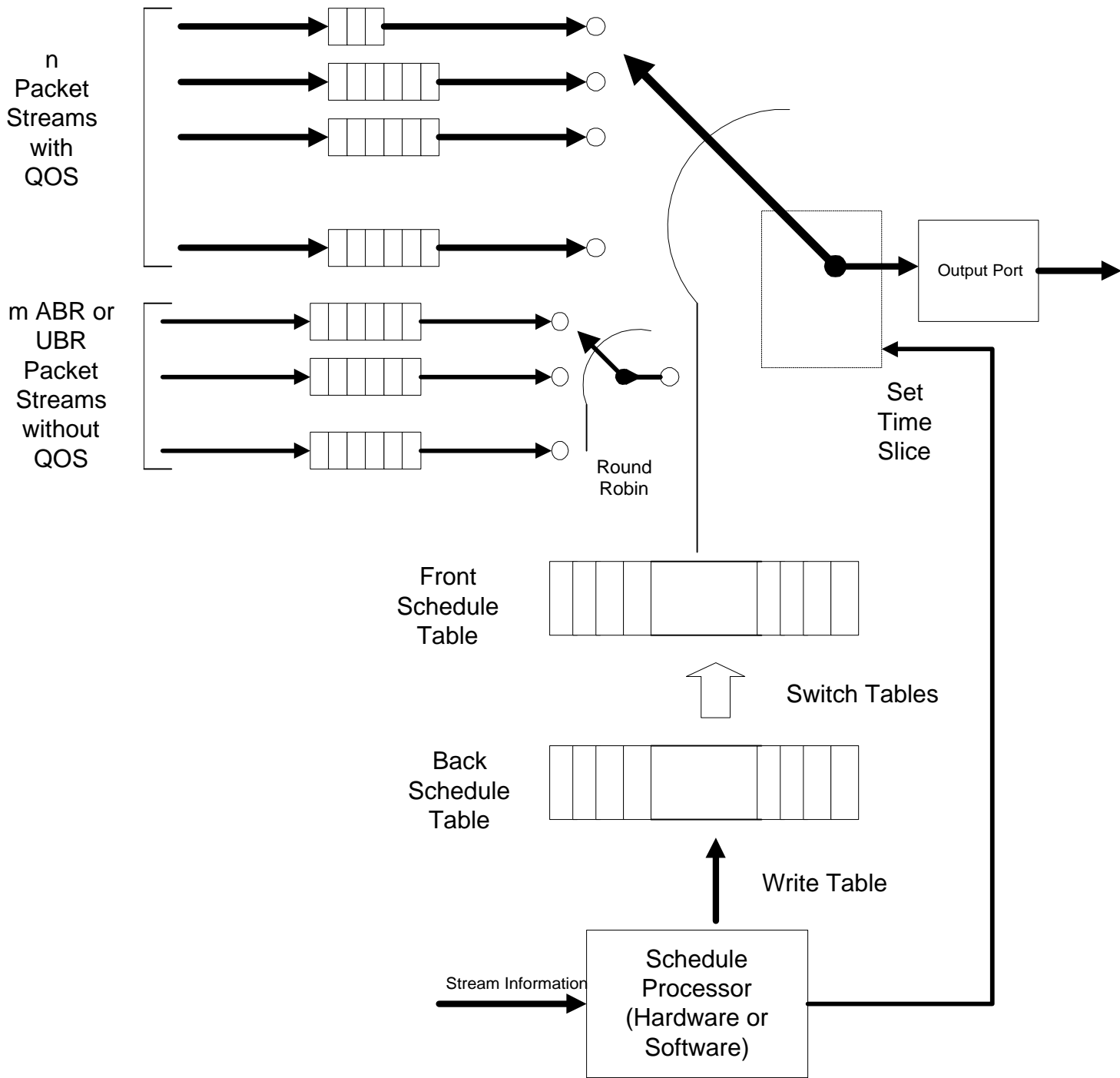


Fig. 3 Complete Scheduler System