Weighted Fair Queueing: a packetized approximation for FFS/GP

- **Packetized Approximation of FFS**
  - **Fact:**
    - **FFS** achieves the **best possible fairness** in **sharing**
  - **Problem with FFS:**
    - **FFS** transmits **one bit** from **each flow** in a **round round fashion**...

      This will **destroy** the **packet structure**

  - **Preserving packet structure while achieving best possible fairness:**
    - **Must** transmit packets ***UNscrabbled***
    - Transmit the packets in the **order of *termination*** achieved in the **FFS** method
  - This is the gist of the **Weighted Fair Queueing** method

  ---

  - **Major contribution:**
    - Develop a **virtual time** method to **compute** the **termination time** of a **packet transmission easily**

- **Nomenclature....**
  - The **algorithm** presented in the paper is called: **Packet-by-packet GPS** in **Parekh's paper**
  - But it is commonly known as the **Weighted Fair Queueing (WFQ)** method.

- **Definitions**
  - **Packet transmission/reception:**
    - **Packet received** = A **packet** has been **received (arrived)** if the ***last bit*** of the **packet has been received (arrived)**
    - **Packet transmitted** = A **packet** is considered **transmitted** if the ***last bit*** of the **packet has transmitted**

    - **Packet finish time** = The **packet finish time** is the time that a packet has been ***transmitted***

      (I.e., the **packet finish time** is the time when the ***last bit*** of the packet **has been transmitted**)

  - **Flows and their reservations:**
    - **C** = the **transmission capacity** of the **communication link**

    - b_f = the **bandwidth guarantee** for **flow** f (= **reservation**)

      (I am using the **notations** used in the **paper**)

    - **Normalized weight (guarantee)** w_f:

      $$w_f = \frac{b_f}{C}$$

    - $l(p_f)$ = the **length (#bytes)** of **packet** p_f of **flow** f

- **Normalized packet length** of a packet p_f:

```
                                 l(pf)
    Normalized Packet Length = ------
                                  C
```

- **The Weight Fair Queueing Scheduler**
  - **Operation** of the **WFQ scheduler** (or **Packet GPS (PGPS)**):
    - **Packet transmission:**
      - **WFQ** transmits **a packet in its *entirety*** before transmitting the **next packet**.
    - **Packet selection for transmission:**
      - **WFQ** selects ***only* packet** that are ***currently* in its queue**
      - **WFQ** selects the **packet** with the **smallest *finish* time** under the **FFS transmission method**
    - **Note:**
      - Once a **packet transmission has *started***, the **packet transmission can*not* be aborted**

      Consequence:

      - It is **possible** that **after** the transmission of a **packet** p_f, a packet p_g arrives that has a **smaller finish time** under **FFS**

      Example:

      - **packet** p_f consists of **100 bytes**
      - **packet** p_g consists of **10 bytes**
      - $w_f = w_g$
      - **Order of arrival:**
        - **Packet** p_g arrives just **after packet** p_f has **started transmission**
      - **Under FFS**: **packet** p_g would **finish *before*** **packet** p_f
      - **However** under **WFQ**:

        Because **packet transmission cannot** be **interrupted**, **packet** p_f will **finish *before*** **packet** p_g

---

- **Operation of the WFQ Scheduler - Example 1**
  - The flows and their weight:
    - flow 1: w_1 = **0.5**
    - flow 2: w_1 = **0.5**
  - **Packet arrival times** and ***Normalized* Packet lengths**:

```
                   Flow 1:                        FLow 2:

            Arrival Time   Packet length     Arrival Time   Packet length
Packet 1:        1              1                 0              3
Packet 2:        2              1                 5              2
```

```
Packet 3:      3           2           9           2
Packet 4:     11           2           -           -
```

- **Recall** the **packet finish times** under the **FFS server**:

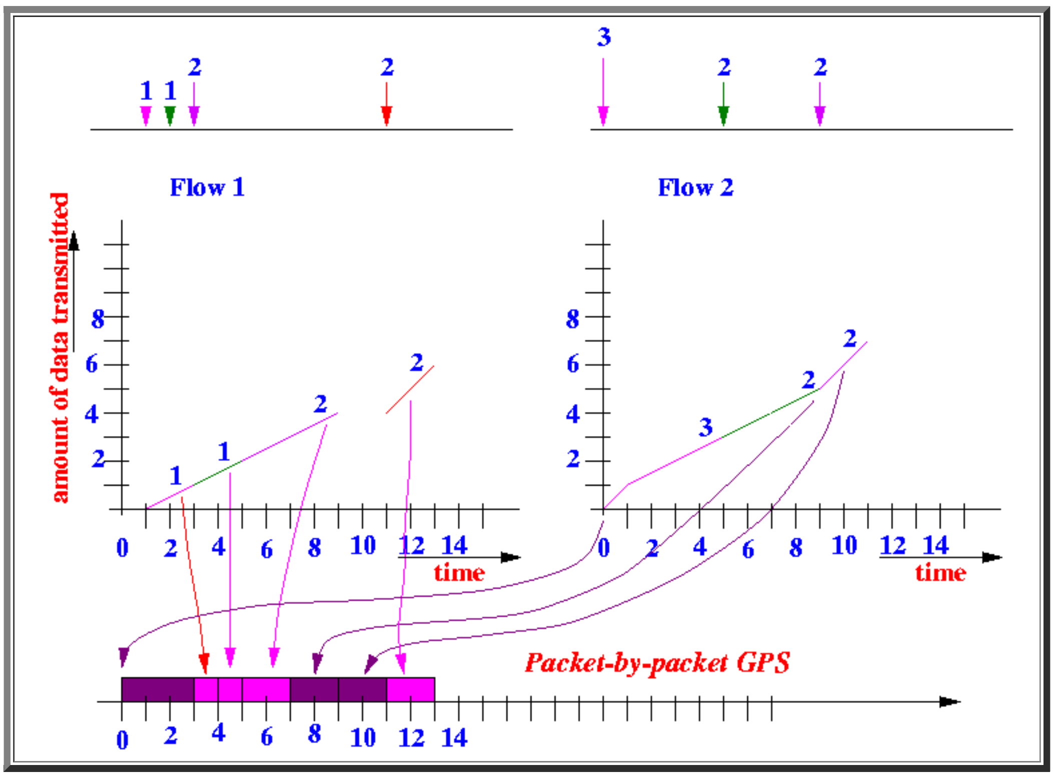
```
                  Flow 1:                           FLow 2:

              Arr Time   Fin Time   length  |   Arr Time   Fin Time   length
Packet 1:        1          3         1     |      0          5         3
Packet 2:        2          5         1     |      5          9         2
Packet 3:        3          9         2     |      9         11         2
Packet 4:       11         13         2     |      -          -
```

- **Scheduling Work sheet**

  (When a packet arrives at time **t**, we assume that it arrives a little later - at **t + ε**)

Time	Event: Arrival (length)	Event: Departure (real finish time)	Packets in queue (FFS finish time) (packet length)	Transmitting (real finis
0	[flow 2, packet 1 (3)]	...	...	[flow 2, packet 1 (3)
1	[flow 1, packet 1 (1)]	...	[flow 1, packet 1 (3) (1)]	...
2	[flow 1, packet 2 (1)]	...	[flow 1, packet 1 (3) (1)] [flow 1, packet 2 (5) (1)]	...
3	[flow 1, packet 3 (2)]	[flow 2, packet 1 (3)]	[flow 1, packet 2 (5) (1)] [flow 1, packet 3 (9) (2)]	[flow 1, packet 1 (4)
4	...	[flow 1, packet 1 (4)]	[flow 1, packet 3 (9) (2)]	[flow 1, packet 2 (5)
5	[flow 2, packet 2 (2)]	[flow 1, packet 2 (5)]	[flow 2, packet 2 (9) (2)]	[flow 1, packet 3 (7)
7	...	[flow 1, packet 3 (7)]	(empty)	[flow 2, packet 2 (9)
9	[flow 2, packet 3 (2)]	[flow 2, packet 2 (9)]	...	[flow 2, packet 3 (11
11	[flow 1, packet 4 (2)]	[flow 2, packet 3 (11)]	...	[flow 1, packet 4 (13
13	...	[flow 1, packet 4 (13)]	...	...

- Summary of the **scheduling order** by the **WFQ scheduler**:



- **Difference in Finish Times between FFS and WFQ servers**

Packet	FFS Finish Time	WFS Finish Time	Difference (WFS - FFS)

Flow 1, Packet 1	3	4	1
Flow 1, Packet 2	5	5	0
Flow 1, Packet 3	9	7	-2
Flow 1, Packet 4	13	13	0
Flow 2, Packet 1	5	3	-2
Flow 2, Packet 2	9	9	0
Flow 2, Packet 3	11	11	0

- **Operation of the WFQ Scheduler - Example 2**
  - Let us work out a second example on how **WFQ** schedule packets for transmission.
  - The weight are changed to following:
    - flow 1: w_1 = **1/3**
    - flow 2: w_1 = **2/3**
  - The **Packet arrival times** and ***Normalized*** **Packet lengths** are **unchanged**:

```
                 Flow 1:                      FLow 2:

          Arrival Time   Packet length    Arrival Time   Packet length
Packet 1:      1              1               0              3
Packet 2:      2              1               5              2
Packet 3:      3              2               9              2
Packet 4:     11              2               -              -
```

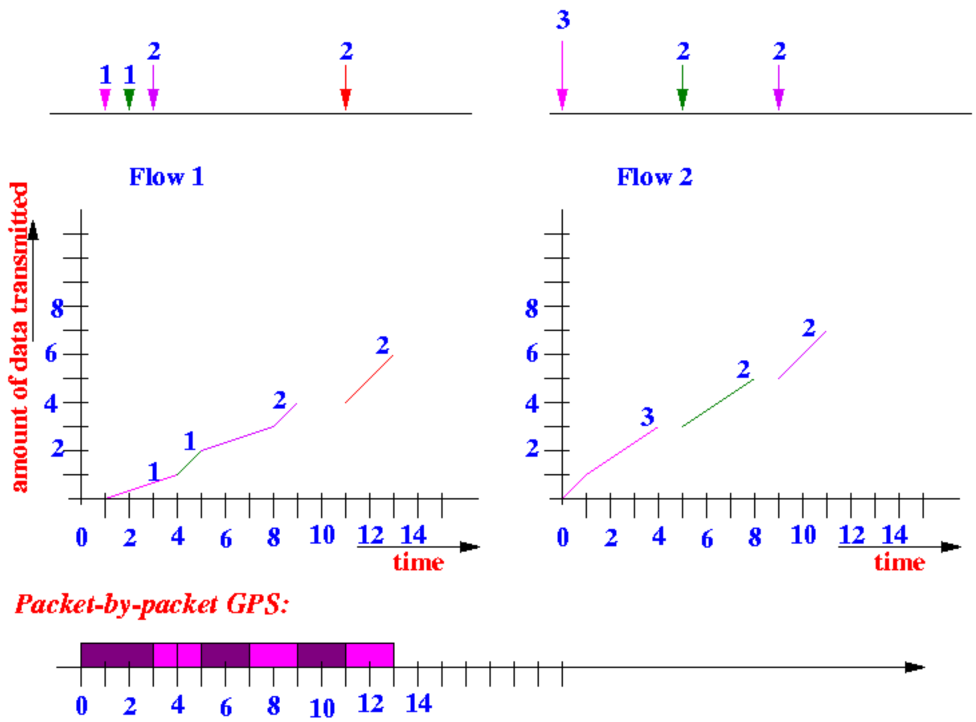
  - We have seen that the **finish times** of each packet when they are transmitted by a **FFS server** are as follows:

```
                 Flow 1:                    FLow 2:

          Arrival Time   Finish Time    Arrival Time   Finish Time
Packet 1:      1              4             0              4
Packet 2:      2              5             5              8
Packet 3:      3              9             9             11
Packet 4:     11             13             -              -
```

  - **Scheduling Work sheet**

Time	Arrival (length)	Departure (real finish time)	Queued Packets (FFS finish time) (packet length)	Scheduled (real finisl
0	[flow 2, packet 1 (3)]	...	[flow 2, packet 1 (4) (3)]	[flow 2, packet 1 (3)]
1	[flow 1, packet 1 (1)]	...	[flow 2, packet 1 (4) (3)] [flow 1, packet 1 (4) (1)]	...
2	[flow 1, packet 2 (1)]	...	[flow 2, packet 1 (4) (3)] [flow 1, packet 1 (4) (1)] [flow 1, packet 2 (5) (1)]	...
3	[flow 1, packet 3 (2)]	[flow 2, packet 1 (3)]	[flow 1, packet 1 (4) (1)] [flow 1, packet 2 (5) (1)] [flow 1, packet 3 (9) (2)]	[flow 1, packet 1 (4)]
4	...	[flow 1, packet 1 (4)]	[flow 1, packet 2 (5) (1)] [flow 1, packet 3 (9) (2)]	[flow 1, packet 2 (5)]
5	[flow 2, packet 2 (2)]	[flow 1, packet 2 (5)]	[flow 2, packet 2 (8) (2)] [flow 1, packet 3 (9) (2)]	[flow 2, packet 2 (7)]
7	...	[flow 2, packet 2 (7)]	[flow 1, packet 3 (9) (2)]	[flow 1, packet 3 (9)]
9	[flow 2, packet 3 (2)]	[flow 1, packet 3 (9)]	[flow 2, packet 3 (11) (2)]	[flow 2, packet 3 (11)
11	[flow 1, packet 4 (2)]	[flow 2, packet 3 (11)]	[flow 1, packet 4 (13) (2)]	[flow 1, packet 4 (13)
13	...	[flow 1, packet 4 (13)]	...	...

  - Summary of the **scheduling order** by the **WFQ scheduler**:



- **Difference in Finish Times between FFS and WFQ servers**

Packet	FFS Finish Time	WFS Finish Time	Difference (WFS - FFS)
Flow 1, Packet 1	4	4	0
Flow 1, Packet 2	5	5	0
Flow 1, Packet 3	9	**9**	0
Flow 1, Packet 4	13	13	0
Flow 2, Packet 1	4	3	-1
Flow 2, Packet 2	8	**7**	-1
Flow 2, Packet 3	11	11	0

- **Summary Conclusions**
  - **What we can learn from the examples:**
    - **WFQ** preserves the **integrity of the packets**
    - **WFQ approximates** the operation of the **FFS** scheduler **very closely**
    - In fact, **Theorems 1 and 2** of **Parekh's paper** states:
      - The ***difference* in the service received** by **any flow** using **WFQ** is:

        $\leq$ **(1 × max. packet length)** ***behind*** the **service** that it **would receive** using **FFS** (**Theorem 2**).

In other words:

- If a **packet** ***p*** finishes service at **time** ***t*** in **FFS**, then **packet** ***p*** will **finish** service at a time ≤ **(*t* + max. packet length)**

Note:

- In the **examples above**, we found that **WFQ** is ≤ **1 time unit** ***behind*** **FFS**.

- The **max length of a packet** is **3 units**

  According the **Theorem 2** of **Parekh's paper**, the **service provided by WFQ** will be **≤ 3 time units** ***behind***

  The results confirms this Theorem.

- **Problems with the FFS-emulation....**
  - In the above **examples**, we saw that to **schedule packets in WFQ**, we must:
    - **compute the finish time** of the **packet** in a **FFS** system

    It is **very complex** to compute the **finish times in FFS**

  - **Major contribution** in **Parekh's paper**:
    - Develop a **simpler scheme** to compute a **(virtual) finish time** that can be used to ***schedule*** **packets in WFQ**

    (This **simpler scheme** is **still** ***quite computational intensive***.)

An (implementable) algorithm for the FFS server

- **Virtual Time**
  - **Virtual time:**
    - The **WFQ** implementation uses a ***virtual*** **time system** which is a very ingeneous **"time keeping mechanism"** for the **FFS** server
    - The **virtual clock** keeps tracks of the **progress** of the **packet transmissions** in the **FFS** server
    - **How to use the virtual clock system:**
      - When a **packet** ***arrives***, the ***virtual*** **finish time** of **that packet** can be **computed** **immediately** using the **virtual time system** !!!

  - **What is *virtual time*:**
    - A **virtual time system** is a **duration measuring system** that is **non-decreasing**
    - The **virtual clock** begins at **virtual time 0 (zero)**

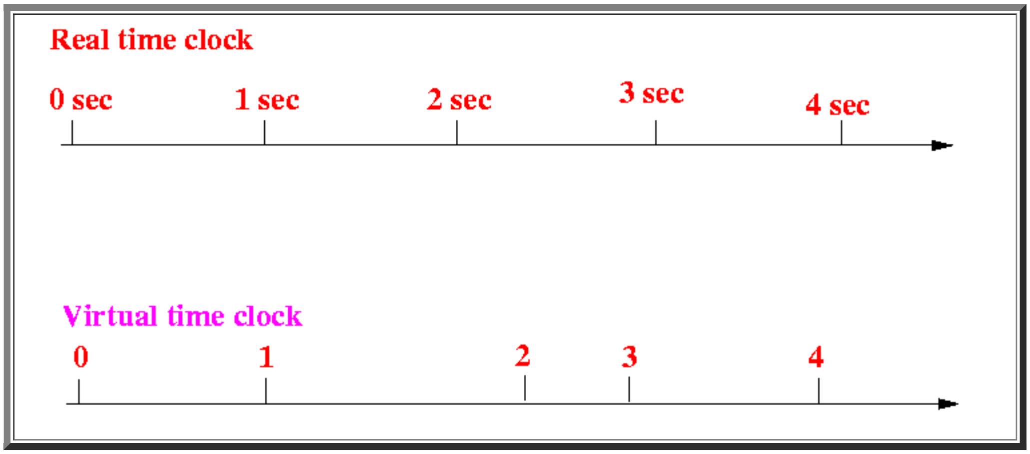
      The **virtual clock** is **non-decreasing** (i.e., you **cannot "go back" in time**)
    - There is a **correspondence function** between **real time (what we experience)** and the **virtual time** (defined in **WFQ**)

      This **correspondence function** will be explained later)

  - **Difference between *real* time and *virtual* time:**
    - In the ***real*** **time system**, the ***rate*** **of progress** (i.e., the ***speed*** **of the real time clock**) is **contant**

- In the ***virtual* time system**, the ***rate* of progress** (i.e., the ***speed* of the virtual time clock**) is **variable** !!!

Example:



- **Reason to create the *virtual time* concept:**

  - The **clock speed** of the **virtual time system** **adapts** to the **service rate of packets**
  - The **clock speed** of the **virtual time system** will **adjust** in such a way that:
    - When a **packet *p* begins transmission**, it will **always *finish*** at the **same *virtual* time**, **regardless** of the **packet arrival patterns** !

  (If you are a fan of **Einstein's relativity theory** (or had the opportunity to travel near the speed of light), you will delight in the following discussion....)

- **Impritant fact to keep in mind:**

  The **speed** of the **virtual clock** is **calibrated** to allow us to compute the ***finish time*** of a **packet** in the **FFS system** **easily**

- **The *Virtual Time* System of WFQ**

  - **Definitions and notations:**

    - t = the **real time** (the **value** of the **wall clock**)
    - $VT(t)$ = the **virtual time value** at **(real) time** t

      $VT(t)$ is a ***function*** that maps the **real time** t to the **virtual time** $VT(t)$

    - **Initialization**: $t = 0$ and $V(0) = 0$

    - w_j = the **weight (reservation)** of **flow** j
    - p_j^i = the i^{th} **packet** of **flow** j

  - **Virtual *finish* time of a packet:**

    - $V(p_j^i)$ = the **virtual finish time** of **packet** p_j^i
    - The **virtual finish time** $V(p_j^i)$ is **also** known as the ***time stamp*** of the packet p_j^i

    - **Fact to be established later:**

- $V(p_j^i)$ = the **virtual time value** when the **packet** p_j^i will **finish its transmission**

- **In other words:**

  - Suppose **packet** p_j^i finishes transmission at **(real) time** y
  - Then: $VT(y) = V(p_j^i)$

- **Furthermore,** we can **compute** $V(p_j^i)$ when **packet** p_j^i **arrives** !!!

  (It's like **predicting the future** :-))

- **Initialization:**

```
V(pj0) = 0,    for every flow j
```

Note:

- There is **no packet** called p_j^0.
- The **first packet** of **flow** j is p_j^1)

- **Virtual flow finish time** of **flow** j = **virtual finish time** of the **last packet** of **flow** j

We will discuss the **time stamp** calculation soon...

- **Components of Parekh's virtual time system:**

  - A **method** to **assign** ***Time Stamps*** **(virtual finish time values)** to an **arriving packet**
  - A **method** to **update the** ***Virtual Clock*** as a **function** of the **real time clock** t

    This **function** will **track packet transmission** and **"simulate"** the **FFS** server

We will **first** look at an **example** that help us **understand Parekh's Virtual CLock scheme**.

- **The** ***speed*** **of the virtual clock when** ***all*** **flows are backlogged**

  - **When** ***all*** **flows are backlogged**

    - $VT(t) = t$

    I.e.: **virtual time** run **at the** ***same*** **rate (speed)** as **real time**

Example:

```
   speed of virtual clock = 1
  -------------------------------->
     V(t) = 1 × t


 |-----------|-----------|----- .....         ----> virtual time (VT(t))
 0           1           2
 ^           ^           ^
 |           |           |
 |           |           |
 |-----------|-----------|----- .....          ----> real time (t)
t=0         t=1         t=2
```

- **Virtual packet length**
  - **Virtual packet length:**
    - The **virtual packet length** determines the **amount of *virtual time*** that is needed to **transmit the packet**
    - **Example:**
      - A **packet** that has a **virtual packet length = *X*** requires ***X* virtual sec** of **virtual time** to transmit the packet
  - We will use an **example** to introduce the concept of **virtual packet length**
  - **Flows:**
    - **Flow f_1**: w_1 **= 0.5**
    - **Flow f_2**: w_2 **= 0.5**
  - **Scenario 1:**
    - At $t = 0$, a **packet** of **length 1** of **flow f_1** arrives
    - At $t = 0$, a **packet** of **length 1** of **flow f_2** arrives

    Note:

    - ***All* flows** are **backlogged**
    - Therefore: $VT(t) = t$

    Service by a *FFS* server:

```
   speed of virtual clock = 1
  -------------------------------->
     V(t) = 1 × t

|-----------|-----------|----- .....   ------> virtual time
0           1           2


                      p1^1 finished
|<---------------------->|
 virtual packet length(p1^1) = 2


                       p2^1 finished
|<---------------------->|
 virtual packet length(p2^1) = 2


|-----------|-----------|----- .....    ------> real time
0           1           2
```

  - **Observation:**
    - In a ***fully loaded* FFS system** (when **every flow** is **backlogged**):
      - A **packet** of **length = 1** from a flow with **weight = 0.5** transmitted (using **FFS**) in **(1/0.5) = 2 *virtual* sec** !!!!

    - **In general: in a *fully loaded* FFS system** (when **every flow** is **backlogged**):
      - A **packet** of **length = L** from a flow with **weight = *w*** transmitted (using **FFS**) in **(L/*w*) sec** !!!!

- **Definition: "virtual packet length"**

```
                              Packet length      L
 Virtual Packet length = ---------------- = ---
                              Weight of flow     w
```

Meaning of "virtual packet length":

- The **virtual packet length (L/w)** = the **amount** of ***virtual*** **time** needed to **transmit a packet** when:
  - **Packet length** = ***L*** **bytes**
  - **weight of the flow** = ***w***

- The **ingenious idea** of **Parekh's scheme** to use **"virtual packet length"** to **track** the **progress of the FFS system**

- **Speed of virtual clock when *not all* flows are backlogged...**
  - **Scenario 2:**
    - At $t = 0$, a **packet** of **length 1** of **flow** f_1 arrives
    - (No packets of **flow** *2*)
  - **Virtual packet lengths:**
    - **Virtual packet length($p_1^{\ 1}$) = 1/(0.5) = 2**

    Service by a *FFS* server:

```
 Virtual packet length of p1¹ = 1/0.5 = 2
      |
      V      p1¹ finished
|<--------->|
            V
|-----------|-----------|-----  .....
0           1           2
```

  - **Observation:**
    - **Virtual packet length of $p_1^{\ 1}$ = 2 (virtual) sec**
    - **Transmission (real) time** of **packet** $p_1^{\ 1}$ = **1 (real) sec**

    - **So:**
      - A **packet** with ***virtual packet length*** = **2 (virtual) sec** was **transmitted** in **1 (real) sec**

    - How can we **make sense** from these **two facts** ?

      (**Fact** = a ***true*** **statement**)

      Hint:

      - **Theory of *relativity*.....**

    - **Answer:**
      - ***speed*** **of the virtual clock** = **2** × ***speed*** **of the real time clock** !!!

      Illustration:

```
     speed of virtual clock = 2
   ------------------------------->
       V(t) = 2 *times; t

V(0)=0                              V(1)=2
 |                                    |
 V                                    V
 |------------------------------------+------
 ^                                    ^
 |                                    |
t=0                                  t=1

   ------------------------------->
       speed of real time clock = 1
```

- **Conclusion:** ***speed* of the virtual clock**

  - When **only flow 1** is **backlogged** ($w_1 = 0.5$), we must use the **virtual function**:

    - $VT(t) = 2 \times t$
      $= t/(0.5)$
      $= t/w_1$

    to **ensure that**:

    - $V(p_1^1) = VT($ ***time when*** p_1^1 ***finished transmission***$)$

- **$64,000 questiuon:**

  - How do we **set** the **speed (rate)** of the **virtual clock** so:

    - The **time stamp** $V(p)$ of a **packet** p is **always equal to (=)** VT(**time when packet** p **finishes transmitting**) ???

- Well, we don't have to figure out this question, **Parekh** did it for his **PhD thesis**...

  We will soon see that: the **the virtual clock function** is:

```
                 t
  VT(t) = ---------------
           Σ       wf
          f is backlogged
```

- **Recall: Components of Parekh's virtual time system:**

  - A **method** to **assign *Time Stamps* (virtual finish time values)** to an **arriving packet**

  - A **method** to **update the *Virtual Clock*** as a **function** of the **real time clock *t***

    This **function** will **track packet transmission** and **"simulate"** the **FFS** server

  We will look at the **time stamp assignment algorithm** first

  Then we will look at **how** to **set the speed** of the **virtual clock**....

- **The WFQ Time Stamp Assignment algorithm**

  - **Recall** that the **meaning** of the ***value* of the time stamp** $V(p)$ of a **packet** p is:

    - $VT(p)$ = the **virtual time** (= the **value of the virtual clock**) when **packet** p **finishes its transmission**

  - **Definitions:**

- **Recall**: w_j = the **(normalized) weight** of **flow** j

---

- p_j^i = the ***last* packet** of **flow** j that has **arrived** at the router
- $V(p_j^i)$ = the **time stamp** of packet p_j^i (= **Virtual Finish Time** of packet p_j^i)

- p_j^{i+1} = the **next (new) packet** of **flow** j that **will arrive**
- A_j^{i+1} = the **arrival time (real time)** of packet p_j^{i+1}
- $VT(A_j^{i+1})$ = the value of the **virtual clock** at time A_j^{i+1} **(real time)**

- $V(p_j^{i+1})$ = the **time stamp** of packet p_j^{i+1} (= **Virtual Finish Time** of packet p_j^{i+1})

Problem:

Find an **expression** for $V(p_j^{i+1})$ such that:

- $V(p_j^{i+1}) = VT(x)$

  where: x = the **time (real)** when packet p_j^{i+1} **finishes transmission**.

Note:

- It **sounds like** that we are trying to **predict the future**,

  and **yes**, we are predicting the future... :-)

- **However**, we are **cheating**:
  - We are **changing** the **speed of the (virtual) clock to make our "prediction" become true** !!!

- **Analogy:**
  - I bet you that I can run a **mile** in **exactly 1 hour**
  - I can ***always* win** the bet ***if*** if have **control** over **how fast** the **clock** runs !!!

- Computing $V(p_j^{i+1})$ for packet p_j^{i+1} - **two cases**:
  - **Case 1:** Packet p_j^i (***last packet*** of flow j) has been **transmitted**
  - **Case 2:** Packet p_j^i (***last packet*** of flow j) has ***not yet* been transmitted**

- **Case 1:** p_j^i has been when p_j^{i+1} arrives

  Graphically depicted:

```
            pj^i departs
             |
             |          pj^i+1 arrives
             |           |
             |           |     len( pj^i+1)/wj
             |           |<----------------------->|
             V           V   virtual length(pj^i+1)
 ------------+-----------+-------------------------> time
                        Aj^i+1
```

```
    -------------+-----------+---------------------------> virtual time
                 VT(A_j^{i+1})                     ^
                                                   |
                                                      len(p_j^{i+1})
                                  V(p_j^i) = VT(A_j^{i+1}) + ----------
                                                              w_j
```

Notes:

1. The **virtual time** at the moment that packet p_j^{i+1} **arrives** is equal to $VT(A_j^{i+1})$
2. The packet p_j^{i+1} received service **immediately** at **virtual time** $VT(A_j^{i+1})$
3. The amount of **virtual time** needed to **transmit** p_j^{i+1} is equal to:

   ```
      len( p_j^{i+1})
     -------------
         w_j
   ```

4. Therefore, the **virtual time** when packet p_j^{i+1} **finishes transmission** is equal to:

   ```
                     len( p_j^{i+1})
     VT(A_j^{i+1}) + -------------
                         w_j
   ```

5. Therefore, the **time stamp** (= **virtual time** when p_j^{i+1} **finishes transmission**) is equal to:

   ```
                                     len( p_j^{i+1})
     V(p_j^{i+1}) = VT(A_j^{i+1}) + -------------                 ....... (1)
                                         w_j
   ```

- **Case 2:** p_j^i has ***not* been transmitted** when p_j^{i+1} arrives

  Graphically depicted:

```
               p_j^{i+1} arrives
                |          p_j^i finishes
                |          p_j^{i+1} begins service
                |           |
                |           |      len( p_j^{i+1})/w_j
                |           |<---------------------->|
                V           V   virtual length(p_j^{i+1})
    -------------+-----------+---------------------------> virtual time
                ^          V(p_j^i)                  |
                |          = Virtual time when p_j^i finish !!
               Now                                   |
                                                     |
                                                       len(p_j^{i+1})
                                  V(p_j^i) = V(p_j^{i+1}) + ----------
                                                             w_j
```

Notes:

1. Packet p_j^{i+1} **starts transmission *immediately* after** packet p_j^i has **finished transmission**.
2. The **virtual time** at the moment that packet p_j^i **finishes transmission** = $V(p_j^i)$
3. The amount of **virtual time** needed to **transmit** p_j^{i+1} is equal to:

   ```
      len( p_j^{i+1})
   ```

```
      -------------
           wj
```

4. Therefore, the **virtual time** when packet p_j^{i+1} **finishes transmission** is equal to:

```
               len( pj^(i+1) )
  V(pj^i)  +  -------------
                    wj
```

5. Therefore, the **time stamp** (= **virtual time** when p_j^{i+1} **finishes transmission**) is equal to:

```
                                len( pj^(i+1) )
  V(pj^(i+1))  =  V(pj^i)  +  -------------                 ....... (2)
                                     wj
```

- **The WFQ *time stamp* assignment algorithm:**

```
                                               len( pj^(i+1) )
  V(pj^(i+1))  =  max( V(Aj^(i+1)), V(pj^i) )  +  -------------           ............ (3)
                                                       wj
```

- **The Speed of the Virtual Clock (= virtual time function $VT(t)$)**
  - We will use **examples** to illustrate **how** the **speed** of the **virtual clock** $VT(t)$ is determined so that:
    - $VT(t)$ will **track the progress of service** in the **FFS system**

  - **Example 1:**
    - **3 flows**:
      - $w_1 = 1/3$
      - $w_2 = 1/3$
      - $w_3 = 1/3$

    - **Packet arrivals** at **time** $t = 0$:
      - len(p_1^1) = 1
      - len(p_2^1) = 1
      - len(p_3^1) = 1

    - **Virtual packet lengths:**
      - virtual len(p_1^1) = 3
      - virtual len(p_2^1) = 3
      - virtual len(p_3^1) = 3

      Meaning:

      - It takes **3 *virtual* sec** to transmit p_1^1
      - It takes **3 *virtual* sec** to transmit p_2^1
      - It takes **3 *virtual* sec** to transmit p_3^1

      Note:

- The packets p_1^1, p_2^1 and p_3^1 can **progress at the *same* time** !!!
- I.e.: **each packet** will take **3 *virtual* sec** to **transmit**

  But: the **packets need *not*** be **transmitted *consecutively***

  (They can be **transmitted *concurrently*** !!!)

- The packets are **serviced as follows**:

```
                                        p1^1 finishes, virtual len(p1^1) = 3
                                        p2^1 finishes, virtual len(p2^1) = 3
                                        p3^1 finishes, virtual len(p3^1) = 3

                                       V(3)=3
  |-----------------------------------|--------------->    virtual time


  |-----------+-----------+-----------+--------------->    real time
 t=0         t=1         t=2         t=3
```

Notes:

- It takes **3 virtual sec** to **transmit** p_1^1, p_2^1 and p_3^1
- p_1^1, p_2^1 and p_3^1 **all started transmission** at ***t* = 0 (real time)**
- p_1^1, p_2^1 and p_3^1 **all finished transmission** at ***t* = 3 (real time)**

- **Therefore:**
  - *VT(3) = 3*

- **Conclusion:**
  - When ***all* flows** are **backlogged**:

```
              t
    VT(t)  = ---
              1
```

- **Example 2:**
  - **3 flows**:
    - $w_1 = 1/3$
    - $w_2 = 1/3$
    - $w_3 = 1/3$

  - **Packet arrivals** at **time** $t = 0$:
    - $\text{len}(p_1^1) = 1$
    - $\text{len}(p_2^1) = 1$

  - **Virtual packet lengths:**
    - **virtual** $\text{len}(p_1^1) = 3$
    - **virtual** $\text{len}(p_2^1) = 3$

- The packets are **serviced as follows**:

```
                        p1^1 finishes, virtual len(p1^1) = 3
                        p2^1 finishes, virtual len(p2^1) = 3

                     V(2)=3
 |-----------------------|--------------------------->    virtual time


 |-----------+-----------+-----------+--------------->    real time
t=0         t=1         t=2         t=3
```

Notes:

- It takes **3 virtual sec** to **transmit** p_1^1, and p_2^1
- p_1^1, and p_2^1 **all started transmission** at $t = 0$ **(real time)**
- p_1^1, and p_2^1 **all finished transmission** at $t = 2$ **(real time)**

- **Therefore:**
  - $VT(2) = 3$

- **Conclusion:**
  - When **flows 1 and 2** are **backlogged**:

$$VT(t) = \frac{t}{(2/3)}$$

- **Example 3:**
  - **3 flows**:
    - $w_1 = 1/3$
    - $w_2 = 1/3$
    - $w_3 = 1/3$

  - **Packet arrivals** at **time** $t = 0$:
    - $\text{len}(p_1^1) = 1$

  - **Virtual packet lengths:**
    - virtual $\text{len}(p_1^1) = 3$

- The packets are **serviced as follows**:

```
            p1^1 finishes, virtual len(p1^1) = 3

         V(1)=3
 |-----------|--------------------------------------->    virtual time


 |-----------+-----------+-----------+--------------->    real time
t=0         t=1         t=2         t=3
```

Notes:

- It takes **3 virtual sec** to **transmit** p_1^1
- p_1^1 **"all" started transmission** at $t = 0$ **(real time)**
- p_1^1 **"all" finished transmission** at $t = 1$ **(real time)**

- **Therefore:**
  - $VT(1) = 3$

- **Conclusion:**
  - When **flows 1** are **backlogged**:

```
                  t
        VT(t) = -----
                (1/3)
```

- **Example 4:**
  - At **time 0**: p_1^1 and p_2^1 arrives.
    len(p_1^1) = 1 and **len(p_2^2) = 1**
  - At **time 1**: p_3^1 arrives.
    len(p_3^1) = 1

  - **Fact: Virtual length** of **each packet** is **still** equal to $1/w = 3$
  - Hence, it will take **still 3 virtual time units** to process **each packet**.

- The packets are services as follows:

```
t=0         t=1         t=2   2.5   t=3
 |           |           |     |     |
 v           v           v     v     v
 |-----------+-----------+-----+-----+---------------> real time
 |<--------->|<--------------->|<--->|
  1/2 P1+P2     1/2 P1+P2+P3    1/2 P3
             ^                 ^     ^
 |<--------->|                 |     |
    1/2 P1   |                 |     |
           1/2 P1            P1 fin  |
           VT=3/2            VT=3    |
             =1.5                    |
             |                       |
             |<--------------->|     |
             |      1/2 P1           |
             |                       |
             |                       |
          P3 starts             P3 fin
                               VT=1.5+3=4.5
             |<--------------------->|
                  P3: 3 virtual sec

 |-----------+-----------+-----+-----+---------------> Virtual time
 0          1.5                 3    4.5
 |<--------->|<--------------->|<--->|
   vt = 1.5 t    vt = 1 * t     vt=3t
      = t/(2/3)     = t/(3/3)     =t/(1/3)
```

Notes:

1. During interval (0, 1), packets **P1** (of flow 1) and **P2** (of flow 2) receive equal service - so half of each packets **P1** and **P2** are transmitted.

   The **virtual packet length** processed is **1.5**, so the virtual time at real time $t = 1$ is equal to $vt = 1.5$ - virtual clock rate is **1.5 units/sec**

2. During interval (1, 2.5), packets **P1**, **P2** and **P3** receive equal service - so half pf each packets **P1** and **P2** are transmitted.

   The **virtual packet length** processed is **1.5**, so the virtual time at real time ***t = 2.5*** is equal to ***vt = 1.5 + 1.5 = 3*** - virtual clock rate is **1 units/sec**

3. During interval (2.5, 3), the last half of packet **P3** is transmitted.

   The **virtual packet length** processed is **1.5**, so the virtual time at real time ***t = 3*** is equal to ***vt = 3+1.5 = 4.5*** - virtual clock rate is **3 units/sec**

- **Conclusion from these examples:**
  - The **speed of the virtual clock depends only** on the ***backlogged* flows in the *FFS* server**

- **Virtual time function of WFQ:**

```
               t
     VT(t) = ------
              S(t)
where

            ----
            \        1
     S(t) =  >      ---
            /        wj
            ----
      All backlogged flows j
      at time real time t
      in a FFS system
```

- **Simulating the FFS using a *digital computer***
  - **Implementation difficulty of the FFS scheduler:**
    - In the **FFS** system, we have to ***continuously*** compute on the progress of the packets

    ---

    - **Digital computers** can **only make discrete time** computations
    - Therefore:
      - The **FFS system cannot** be **implemented** of a **digital computer**
      - You would need an **analog computer** to simulate the **FFS system**

  - **Contribution by Parekh:**
    - **Fact:**
      - The **speed of the virtual clock** depends ***only*** on the **backlogged flows** in the **FFS server**

    - **When can a flow change its state between *non-backlogged* and *backlogged***:
      - When a **(new) packet arrives**
      - When a **packet finishes transmission** in the **FSS server**

    - **Therefore:** the **speed of the virtual clock** will ***only*** **change** at these **moments**:
      - When a **(new) packet arrives**
      - When a **packet finishes transmission** in the **FSS server**

  - **Parekh's** observations made it possible to **examine a *finite* number** of **moments in time** and **compute** the **progress** of the **FFS server**

- **The WFQ scheduling algorithm**
  - The **implementation** of an **simulation of the FFS scheduler** require us to ***monitor*** these types of **events** :
    - e_1 = the **event** that a **packet arrives**
      - **Because:** If an **arrival** causes some **flow** to become **backlogged** (i.e., **first packet**), then the **rate of the virtual clock** will **change**
    - e_2 = the **event** that a **packet finishes** in the **FFS server**
      - **Because:** If a **packet finishes transmission** in the **FFS server** causes some **flow** to become ***not* backlogged** (i.e., **last packet** transmitted), then the **rate of the virtual clock** will **change**
  - The **WFQ scheduler** will ***use* the FFS simulation** to ***schedule* the next packet** for **transmission**

    So we must **also *monitor*** these types of **events in *real time***:
    - e_3 = the **event** that a **packet finishes transmission** in **real time**
      - **Because:**
        - The **WFQ scheduler** transmits the **packet atomically**
        - When a **packet finishes transmission** in **real time**, the **next packet** (with the **smaller *virtual finish time***) must be **selected** to **start transmission**
  - **Definitions:**
    - $t(e_1)$ = the **earliest *real* time** of a **packet arrival event**
    - $t(e_2)$ = the **earliest *real* time** of a **packet departure event** in the **FFS server**
    - $t(e_3)$ = the **earliest *real* time** of a **packet departure event** in the **WFQ server**

    Notation:

    t_{next} = **min(** $t(e_1)$, $t(e_2)$, $t(e_3)$)
      - t_{next} = the **"next event time"**
        - the **next *real* time moment** that **WFQ scheduler** must perform some **action**
  - **Actions performed by the *WFQ scheduler* for a specific event:**
    - **When the next packet p_j^{i+1} arrives:**

      Notations:
      - Let $A(p_j^{i+1})$ = the ***arrival time (real time)*** of the packet p_j^{i+1}
      - Let p_j^i = the ***last* packet** of **flow *j*** that arrived to the **WFQ scheduler**
      - Let $V(p_j^i)$ = the ***time stamp*** assigned to the packet p_j^i
        = the **virtual time** that packet p_j^i **finishes transmission** !!

      1. **Assign** a ***time stamp* (virtual finish time)** to packet p_j^{i+1} :

$$V(p_j^{i+1}) = \max(\ V(A_j^{i+1}),\ V(p_j^i)\ ) + \frac{\text{len}(\ p_j^{i+1})}{w_j}$$

2. If the **WFQ scheduler** is **idle**, then:

   - **Transmit** packet p_j^{i+1}

3. **Re-compute** the **speed** of the **virtual clock**:

```
                              ----
                              \        1
Speed of virtual clock =       >      ---
                              /        wj
                              ----
                          All active flows j
                          at time real time t
                          in a FFS system
```

4. ***Re-compute:***

   - $t(e_2)$ = the **earliest *real* time** of a **packet departure event** in the **FFS server** (using the **updated speed** of the **virtual clock**).

- **When the (simulated) FFS server has finished transmitting a packet:**

1. **Re-compute** the **speed** of the **virtual clock**:

```
                              ----
                              \        1
Speed of virtual clock =       >      ---
                              /        wj
                              ----
                          All active flows j
                          at time real time t
                          in a FFS system
```

2. ***Re-compute:***

   - $t(e_2)$ = the **earliest *real* time** of a **packet departure event** in the **FFS server** (using the **updated speed** of the **virtual clock**).

- **When the WFQ server has finished transmitting a packet(atomically):**

1. Determine the packet ***p*** where ***V(p) = minimum* virtual finish time** of **all packets in the queue**
2. **Transmit** packet ***p***
3. ***Re-compute:***

   - $t(e_2)$ = ***now (real time)*** + **len(*p*)**

     len(*p*) = ***normalized* packet length** (amount of **time (sec)** needed to **transmit** packet ***p***)

- **WFQ Psuedo code**
  - **Variable utilization:**
    - **t** = the **current real time (clock)**
    - **vt** = the **current virtual time (clock)**
    - **prev_t** = the **last event time (in real time)**
    - **prev_vt** = the **last event time (in virtual time)**
    - **S** = the **speed of the virtual time clock**

    - **PacketQ** = the **queue of packets** in the **WFQ scheduler**

      The **packets** are **ordered** by their **(virtual) time stamp**

      The **first packet** in **PacketQ** is being **transmitted**
    - **PacketQ.remainingTransmitTime** = the **remaining (real) time** needed to **transmit** the **first packet** in **PacketQ**

    - **lastTimeStamp[flowID]** = the **last (virtual) time stamp** assigned to a **packet** of ***flow*** **flowID**
    - **RemainingVirLength[j]** = the **remaining (virtual) packet length** of the ***earliest*** **packet** of **flow *j*** (whcih is in service at the **FFS server**)

  - **Initialization:**

```
prev_t  =  t = 0;          // Clock
prev_vt = vt = 0;          // Virtual clcok

for ( j = 0; j < nFlows; j++ )
{
   lastTimeStamp[j] = 0;
   RemainingVirLength[j] = 0;
}
```

  - **WFQ scheduling algorithm:** (there are **2 server codes** (**FFS** and **WFQ**) in the algorithm !)

```
/* --------------------------------------------------------
   Help function: Computing the rate of the virtual clock
   -------------------------------------------------------- */
double compute_VT_Rate()
{
   double r = 0.0;

   for (int i = 0; i < nFlows; i++)
      if ( Flow[i].size() > 0 )
         r += w[i];

   if ( r > 0 )
      return(1/r);
   else
      return(0.0);
}
```

```
/* --------------------------------------------------------
   Help function: minimum of 3 values
   -------------------------------------------------------- */
double min3(double t1, double t2, double t3)
{
   if ( t1 ≤ t2 && t1 ≤ t3 )
      return t1;
   else if ( t2 ≤ t1 && t2 ≤ t3 )
      return t2;
   else
      return t3;
}
```

```
while ( true )
{
   /* --------------------------------
```

```
      Record the previous action time
      -------------------------------- */
   prev_t = t;
   prev_vt = vt;

   /* ==========================================
      Compute the next action time (real time)
      ========================================== */

   /* ----------------------------------------
      Get t(e1) = next packet arrival time
      ---------------------------------------- */
   Let t_arrival = arrival time of next packet;

   /* ------------------------------------------------------
      Compute t(e2) = next packet finish time in FFS server
      ------------------------------------------------------ */
   vt_rate = compute_VT_Rate();

   for ( i = 0; i < nFlows; i++ )
   {
      if ( Flow[i].get(0).RemainingVirlength > 0 )
         t_VirtualFinTime[i] = Flow[i].get(0).RemainingVirlength/vt_rate;
   }
   t_FFS_finish = min( t_VirtualFinTime[i = 0..(nFlows-1)] );

   /* ------------------------------------------------------
      Compute t(e3) = next packet finish time in WFQ server
      ------------------------------------------------------ */
   if  ( PacketQ != null )
      t_departure = t + PacketQ.remainingTransmitTime;
   else
      t_departure = MAX_DOUBLE;

   /* ============================================================
      Update progress in the FFS and the WFS
      ============================================================ */
   /* ---------------------------------
      Advance (real time) clock
      --------------------------------- */
   t = min3(t_arrival, t_departure, t_FFS_finish) );
   vt = prev_vt + vt_rate * (t - prev_t);

   /* -------------------------------------
      Update progress (from prev_t --> t)
      ------------------------------------- */
   for ( i = 0; i < nFlows; i++ )
   {
      if ( Flow[i].get(0).RemainingVirlength > 0 )
          Flow[i].get(0).RemainingVirlength =
                        Flow[i].get(0).RemainingVirlength
                        - vt_rate*(t-prev_t);
   }

   if ( PacketQ != null )
   {
      PacketQ.remainingTransitTime = PacketQ.remainingTransitTime
                                     - (t - prev_t);
   }



   /* ---------------------------------
      Process event (the WFQ algorithm)
      --------------------------------- */
   if ( t_arrival == t )
   {  /* --------------------------
         Handle an arrival event
         -------------------------- */
      packet = new Packet();
      flowID = flow ID of arriving packet'

      packet.packetLen = length of packet;
      packet.remainingVirLength = packetLen/w[flowID];
      packet.TimeStamp = max( vt, lastTimeStamp[flowID])
                         + packet.remainingTransitTime;

      lastTimeStamp[flowID] = packet.TimeStamp;

      insertOrdered( packet, PacketQ );
      insertFIFO( packet, Flow[i] );
   }
   else if ( t_FFS_finish == t )
   {  /* -----------------------------------
         Handle a departure in the FFS server
         ----------------------------------- */
      for ( i = 0; i < nFlows; i++ )
      {
         if ( Flow[i].size() > 0 )
         {
            if ( Flow[i].get(0).RemainingVirlength < 0.00001 )
```

```
                {
                    Flow[i].remove(0);
                }
             }
          }
       }
       else
       {  /* ------------------------------------
             Handle a departure in the WFQ server
             ------------------------------------ */
          PacketQ = PacketQ.deleteHead();

          if ( PacketQ != null )
          {
             PacketQ.remainingTransmitTime = PacketQ.packetLen;
          }
       }
    }
```

- **WFQ Scheduling - Example**
  - Consider the **Example 4** from above:

    - At **time 0**: p_1^1 and p_2^1 arrives.
      len(p_1^1) = 1 and **len(p_2^2) = 1**
    - At **time 1**: p_3^1 arrives.
      len(p_3^1) = 1

    - **Fact: Virtual length** of **each packet** is **still** equal to ***1/w = 3***
    - Hence, it will take **still 3 virtual time units** to process **each packet**.

  - **Worked out example:**

```
Initialization:

  prev_t = 0   prev_vt = 0
  t = 0        vt = 0

      FFS server                          WFQ server
      ----------                          -----------
      1:                          PacketQ:
      2:
      3:

 vt_rate = ?

-----------------------------------------------------------

Event 1:     t = 0     p1^1 arrives

  prev_t = 0   prev_vt = 0
  t = 0        vt = 0

     (No progress, because t == prev_t)

      FFS server                              WFQ server
      ----------                              -----------
      1: p1^1(packetLen=1              PacketQ: p1^1(remaingTransmitTime=1)
            remVirLenghth=3
            TimeStamp=3)
      2: ---
      3: ---

vt_rate = 3/1
=====================

   t_arrival = 0
   t_FFS_finish = min( (3.0/3.0) ) = 1.0
   t_departure = 1

======================

-----------------------------------------------------------

Event 2:     t = 0     p2^1 arrives

  prev_t = 0   prev_vt = 0
```

```
  t = 0        vt = 0

     (No progress, because t == prev_t)

      FFS server                              WFQ server
      ----------                              -----------
      1: p1^1(packetLen=1              PacketQ: p1^1(remaingTransmitTime=1) p2^1
            remVirLenghth=3
            TimeStamp=3)
      2: p2^1(packetLen=1
            remVirLenghth=3
            TimeStamp=3)
      3: ---

vt_rate = 3/2
=====================

   t_arrival = 1
   t_FFS_finish = min( (3.0/1.5), (3.0/1.5) ) = 1.5
   t_departure = 1

======================
```

```
Event 3:    t = 1    p3^1 arrives

  prev_t = 0   prev_vt = 0
  t      = 1   vt      = 1.5

      FFS server                              WFQ server
      ----------                              -----------
      1: p1^1(packetLen=1              PacketQ: p1^1(remaingTransmitTime=0) p2^1 p3^1
            remVirLenghth=1.5
            TimeStamp=3)
      2: p2^1(packetLen=1
            remVirLenghth=1.5
            TimeStamp=3)
      3: p3^1(packetLen=1
            remVirLenghth=3
            TimeStamp=4.5)    = [1.5 + 3]

vt_rate = 3/3
=====================

   t_arrival = ?
   t_FFS_finish = min( (1.5/1.0), (1.5/1.0), 3.0/1.0 ) = 1.5
   t_departure = 0

======================
```

```
Event 4:    t = 1    p1^1 finishes in WFQ server

  prev_t = 1   prev_vt = 1
  t      = 1   vt      = 1.5

     (No progress, because t == prev_t)

      FFS server                              WFQ server
      ----------                              -----------
      1: p1^1(packetLen=1              PacketQ: p2^1(remaingTransmitTime=1) p3^1
            remVirLenghth=1.5
            TimeStamp=3)
      2: p2^1(packetLen=1
            remVirLenghth=1.5
            TimeStamp=3)
      3: p3^1(packetLen=1
            remVirLenghth=3
            TimeStamp=4.5)    = [1.5 + 3]

vt_rate = 3/3
=====================

   t_arrival = ?
   t_FFS_finish = min( (1.5/1.0), (1.5/1.0), 3.0/1.0 ) = 1.5
   t_departure = 1

======================
```

```
Event 5:    t = 2    p2^1 finishes in WFQ server

  prev_t = 1   prev_vt = 1
  t      = 2   vt      = 2.5

      FFS server                              WFQ server
      ----------                              -----------
```

1: p_1^1(packetLen=1 PacketQ: p_3^1(remaingTransmitTime=1)
remVirLenghth=0.5
TimeStamp=3)
2: p_2^1(packetLen=1
remVirLenghth=0.5
TimeStamp=3)
3: p_3^1(packetLen=1
remVirLenghth=2
TimeStamp=4.5)

vt_rate = 3/3
=====================

t_arrival = ?
t_FFS_finish = min((0.5/1.0), (0.5/1.0), 2.0/1.0) = 0.5
t_departure = 1

======================

Event 6: t = 2.5 p_1^1 finishes in FFS server

prev_t = 2 prev_vt = 2
t = 2.5 vt = 3

FFS server WFQ server
---------- -----------
1: ~~p_1^1(packetLen=1~~ PacketQ: p_3^1(remaingTransmitTime=0.5)
~~remVirLenghth=0.0~~
~~TimeStamp=3)~~
2: p_2^1(packetLen=1
remVirLenghth=0.0
TimeStamp=3)
3: p_3^1(packetLen=1
remVirLenghth=1.5
TimeStamp=4.5)

vt_rate = 3/2
=====================

t_arrival = ?
t_FFS_finish = min((0.0/1.5), 1.5/1.5) = 0.0
t_departure = 0.5

======================

Event 7: t = 2.5 p_2^1 finishes in FFS server

prev_t = 2.5 prev_vt = 3
t = 2.5 vt = 3

(No progress, because t == prev_t)

FFS server WFQ server
---------- -----------
1: ~~p_1^1(packetLen=1~~ PacketQ: p_3^1(remaingTransmitTime=0.5)
~~remVirLenghth=0.0~~
~~TimeStamp=3)~~
2: ~~p_2^1(packetLen=1~~
~~remVirLenghth=0.0~~
~~TimeStamp=3)~~
3: p_3^1(packetLen=1
remVirLenghth=1.5
TimeStamp=4.5)

vt_rate = 3/1
=====================

t_arrival = ?
t_FFS_finish = min(1.5/3.0) = 0.5
t_departure = 0.5

======================

Event 8: t = 3 p_3^1 finishes in FFS server

prev_t = 2.5 prev_vt = 3
t = 3 vt = 4.5 (= 3 + 0.5×3)

FFS server WFQ server
---------- -----------
1: ~~p_1^1(packetLen=1~~ PacketQ: p_3^1(remaingTransmitTime=0.0)
~~remVirLenghth=0.0~~
~~TimeStamp=3)~~

```
      2: p2^1 (packetLen=1
              remVirLenghth=0.0
              TimeStamp=3)
      3: p3^1 (packetLen=1
              remVirLenghth=0.0
              TimeStamp=4.5)

vt_rate = 3/1
=====================

   t_arrival = ?
   t_FFS_finish = min( ? ) = ?
   t_departure = 0.0

======================
```

```
Event 8:    t = 3    p3^1 finishes in WFQ server

  prev_t = 3     prev_vt = 4.5
  t      = 3     vt      = 4.5

     (No progress, because t == prev_t)

      FFS server                          WFQ server
      ----------                          -----------
      1: p1^1 (packetLen=1              PacketQ: ---
              remVirLenghth=0.0
              TimeStamp=3)
      2: p2^1 (packetLen=1
              remVirLenghth=0.0
              TimeStamp=3)
      3: p3^1 (packetLen=1
              remVirLenghth=0.0
              TimeStamp=4.5)

vt_rate = 3/1
=====================

   t_arrival = ?
   t_FFS_finish = min( ? ) = ?
   t_departure = 0.0

======================
```

Done

- The **virtual clock** is **kept *precisely*** !!!

- **Example Program:** (Demo above code) ***Example***
  - Prog file: click here

    (I need to clean up the code for this implementation... It works, but the code is **ugly**)

- **Sample outputs:**
  - Input set 1: **click here**
  - Input set 2: **click here**
- **Sample outputs:**

```
Arrivals:

 T = 0.0, flow 2, length 3.0
 T = 1.0, flow 1, length 1.0
 T = 2.0, flow 1, length 1.0
 T = 3.0, flow 1, length 1.0
 T = 5.0, flow 2, length 2.0
 T = 9.0, flow 2, length 2.0
 T = 11.0, flow 1, length 2.0


Begin WFQ computation




==========================================
last_update_time = 0.0
(1)  Current packet Queue:
   -- empty
(2)  Fluid Flow Server State:
   -- empty
------------------------------
BEGIN processing
t = 0.0 , vt = 0.0
+++ Next packet arrival time   = 0.0   (flow ID = -1, packet len = -1.0)
+++ Next packet departure time = 9999999.0
+++ Next FFS service end time  = 9999999.0

------------------------------------------
----- Find out Next Event in WFQ ---------
Next event: packet arrival
------------------------------------------
***** Processing packet arrival

vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
END processing: result
```

- **How well does WFQ approximate FFS ?**
  - **Theorem**

    - Let p be an **arbitrary packet** and let t^{fin}_p be the **finish time of packet** p in the **Fluid Flow Server**
    - Then **finish time of packet** p in the **WFQ server** at **less than or equal to:**

      $$t^{fin}_p + t_{MAX}$$

      where t_{MAX} is the **time** needed to transmit the **largest packet in the system**

  - BTW, that's the **best** that you can do, because if your packet arrives just one nano-second behind the largest packet in the system, you will have to wait until that packet is sent before your packet can be transmitted.

    In this worst case scenario, your service will be t_{MAX} behind schedule...
