Algorithm to Implement the WFQ Scheduler with Changing Weights. Operation Details of WFQ over ns-2. Modification of WFQ over ns-2 for operation with dynamic weights. General Operation and structure of ns-2. Details of routines and variables of ns-2 relative to the operation of queues.

Technical Report.

By ALFREDO PIERO MATEOS PAPIS.

UAM – Cuajimalpa.

Table of Contents.

1 El Despachador GPS..... 2

1.1 Fundamentos..... 2

1.2 Interacción con el Despachador PGPS..... 4

2 Despachador GPS con Pesos que Cambian según la Longitud de la Cola..... 11

3 Algoritmo Propuesto para Sistema de Despacho GPS con Pesos que Cambian Según la Longitud de las Colas..... 12

3.1 Algoritmo con Detalles de Implantación..... 12

3.1.1 Caso de la Llegada de un Paquete. Sistema de Despacho GPS. 13

3.1.2 Caso de la Salida de un Paquete. Sistema de Despacho GPS. 17

Apéndice 1. Operación de ns..... 22

Apéndice 2. Detalles sobre la Operación de ns con el despachador WFQ..... 26

Apéndice 3. Detalles de Rutinas y Variables de ns Relativas a la Operación de Colas (antes de la modificación realizada por Alfredo Mateos)..... 33

Apéndice 4. Modificaciones al Simulador ns para la Operación de WFQ con Pesos que tienen Cambio Dinámico..... 49

Estructura de las Clases en la Programación del Simulador ns..... 55

Apéndice 5. Configuraciones Adicionales..... 56

Despachador.

Un despachador es un elemento que existe en una interfaz de salida de un enrutador cuando la misma tiene varias colas de espera (las colas tienen paquetes). Ya que la interfaz solo tiene un enlace de salida, se pueden enviar (también se dice: despachar, atender o servir) un paquete a la vez. El despachador selecciona qué paquete va a ser el próximo a enviarse por dicho enlace (el que también se refiere como el enlace de salida del despachador). Un despachador muy conocido es el "Round Robin". Mientras el despachador está enviando un paquete por el enlace de salida se dice que el despachador está "atendiendo" el paquete.

1 El Despachador GPS.

1.1 Fundamentos.

El despachador GPS (General Processor Sharing) es un tipo "ideal" de despachador que sirve como referencia para análisis y comparación (referirse a [1] para más detalles).

En el despachador GPS se considera que el tráfico no está hecho de paquetes sino de un líquido que puede fluir por tuberías. El enlace de salida sería sustituido por N tuberías por donde pasaría el tráfico de cada uno de las colas. El tráfico de llegada al despachador se seleccionará y distribuirá para despacho, según su tipo, de entre los N tipos que hay, en las N tuberías de salida.

Si alguna cola quedase vacía, en algún intervalo de tiempo, en el mismo las demás colas pueden aprovechar la capacidad de la tubería no utilizada, de tal forma que nunca ninguna tubería quedará en desuso mientras haya tráfico en espera de ser atendido en el despachador. En esta forma el despachador no desperdicia nunca sus recursos de transmisión disponibles mientras tenga tráfico a enviar.

Cuando un despachador tiene esta característica de no desperdiciar sus recursos de transmisión se dice que el mismo es "Conservador de Trabajo" o "Work-Conserving". En otras palabras, un despachador es Conservador de Trabajo cuando el mismo envía (sirve) a máxima tasa mientras el mismo tenga tráfico a atender.

El despachador GPS está caracterizado por valores positivos reales ϕ_1, ϕ_2, ..., ϕ_N, que son los pesos de sus N colas. Estos pesos se usan para establecer la mínima tasa con que se atenderá al tráfico de una tubería en cualquier intervalo en donde haya tráfico en espera para dicha tubería (se dice mínima tasa porque es posible que la tubería pudiese aprovechar en dicho intervalo las posibles capacidades no utilizadas de otras tuberías).

Dígase que la tasa total de salida del despachador es R (en el mundo de las redes de datos a esta cantidad se le refiere también como "ancho de banda").

Desde luego GPS no es realizable, pero sirve como fundamento para la operación de otro despachador llamado PGPS (Packet-by-Packet GPS[1]) que sí es realizable pero que su operación

[1] Existe una propuesta con la misma operación a PGPS llamada WFQ (Weighted Fair Queueing)

causa mucho trabajo de cálculo y no se considera viable en ambientes de operación productivos (en donde los enrutadores requieren atender a muchos miles de paquetes por segundo) sino para objetivos de estudio, por ejemplo con simuladores.

Operación de GPS.

Sea $S_i(\tau, t)$ la cantidad de tráfico de la cola *i* que será atendida en el lapso $(\tau, t]$. Si la cola *i* tiene continuamente tráfico por atender en dicho lapso[2], entonces se debe cumplir, según la operación de GPS que:

$$\frac{S_i(\tau,t)}{S_j(\tau,t)} \geq \frac{\phi_i}{\phi_j}, j = 1, 2, ..., N. \quad \textbf{(1)}$$

Continuando con la consideración de que la cola *i* está continuamente ocupada en $(\tau, t]$ (es decir en el conjunto de colas al menos una de ellas está continuamente ocupada), entonces se debe cumplir que

$$\sum_{\forall j} S_j(\tau,t) = (t-\tau)R$$

Donde *R* es el "ancho de banda" del enlace de salida del despachador. Lo anterior quiere decir que el tráfico atendido en el lapso $(\tau, t]$ es tan grande como el tráfico máximo que puede atender el despachador en ese lapso, que es igual a $(t-\tau)R$.

La ecuación (1) se debe cumplir. El despeje de la misma sería $\frac{\phi_j}{\phi_i} S_i(\tau,t) \geq S_j(\tau,t)$, $j = 1, 2, ..., N$. Al sustituir este despeje en la ecuación anterior se obtiene

$$\frac{S_i(\tau,t)}{\phi_i}\sum_{\forall j}\phi_j \geq \sum_{\forall j} S_j(\tau,t) = (t-\tau)R\phi_j$$

y despejando se obtiene

$$\frac{S_i(\tau,t)}{(t-\tau)} \geq \frac{\phi_i}{\sum_{\forall j}\phi_j} R$$

Donde $\frac{S_i(\tau,t)}{(t-\tau)}$ es la tasa de atención a la cola $S_i(\tau,t)$ durante el lapso $(t-\tau]$. De lo anterior se deriva que la tasa garantizada mínima, $R_i(\tau,t)$, para la cola *i*, durante dicho lapso, es:

[2] Cuando hay tráfico por atender en la cola i se utiliza un término en inglés para indicar que la cola está retrasada en su trabajo por hacer. En inglés se diría que la cola i está "Backlogged".

$$R_i(\tau,t) = \frac{\phi_i}{\sum_{\forall j}\phi_j} R \qquad \textbf{(2)}$$

Pudiendo la tasa para la cola *i* ser mayor cuando en una o más de las otras colas no hay trabajo por hacer.

Considérese que hay un intervalo (τ_1, t_1] tan pequeño como se requiera para que las colas mantengan una tónica, es decir, o tienen trabajo por hacer continuamente, o no tienen trabajo por hacer en absoluto. Entonces, la tasa que tendrá la cola *i* será.

$$R_i(\tau,t) = \begin{cases} \dfrac{\phi_i(\tau,t)}{\sum\limits_{\phi_j \in B(\tau,t)} \phi_j} R, & \phi_i \in B(\tau,t) \\ 0, & \phi_i \notin B(\tau,t) \end{cases} \qquad \textbf{(3)}$$

Donde $B(\tau,t)$ es el conjunto de colas con trabajo por hacer en el intervalo (τ,t).

Los detalles sobre la operación del despachador WFQ en las rutinas C++ del simulador ns se dan en el Apéndice 2.

Llámese evento al inicio de atención de un paquete o a la finalización de atención de un paquete, por parte del despachador GPS. Cuando el despachador está enviando a un paquete de una cola dada se dice que el despachador está atendiendo al paquete, y a su vez a la cola, y también se dice que el despachador está teniendo una sesión con dicha cola. Mientras un paquete de una cola dada está siendo atendido por el despachador se considera que el paquete sigue estando en la cola, y mientras haya al menos un paquete en una cola se dice que la misma está ocupada. Un despachador GPS atenderá a los paquetes de cada cola en una forma FIFO (primero que entra primero en ser servido) y siempre que haya al menos un paquete en una cola el despachador estará atendiendo a la cola (al primer paquete en turno en la cola), esto quiere decir que el despachador FIFO es un despachador *Conservador de Energía* (siempre trabaja al máximo de su capacidad cuando hay al menos un paquete en espera en al menos una cola).

Cuando se finaliza la atención de un paquete, el mismo ha salido completamente del despachador y de la cola en donde estaba. Justo cuando se finaliza el envío de un paquete de una cola dada se puede iniciar el envío del siguiente paquete en espera en esa cola (si lo hay).

1.2 Interacción con el Despachador PGPS.

En [1] se presentan y explican las características y la operación de los despachadores GPS y PGPS, y se ofrece detalle sobre la relación de tiempo en que el despachador PGPS difiere con relación al despachador GPS, para la atención de los paquetes.

Un despachador PGPS se basa en la operación de un despachador GPS, pero el primero solo puede atender a un paquete a la vez, pues es un despachador *real*, no de fluido.

Divídase el tiempo en intervalos entre eventos. Cada intervalo concluye cuando hay un evento, quedando dicho evento situado al inicio del siguiente intervalo.

Sea t_{b-1} el tiempo en que ocurre un evento, y sea t_b el tiempo en que ocurre el siguiente evento. Entonces el conjunto de colas ocupadas (de sesiones activas) está fijo durante ese intervalo $[t_{b-1}, t_b)$, que se denota como B_b. De un intervalo a otro el conjunto de colas ocupadas puede cambiar o permanecer igual.

Denótese el tiempo en que un paquete #*k* llega a la cola #*i* como a_i^k, y a la longitud de ese paquete como L_i^k, y al paquete mismo p_i^k.

Durante el tiempo de atención de dicho paquete podrían llegar cero, uno o más paquetes. El análisis en que no llegase ningún paquete sería el más sencillo, el cual no se hará, sino se hará uno suponiendo que llega un paquete solamente durante dicho tiempo de atención (la generalización al caso en que llegase más de un paquete sería sencilla).

Considérese que t_1 es el tiempo de inicio de operación del sistema de colas (se toma entonces como que $t_1 = 0$), y que la atención al paquete p_i^k se da en un lapso de tres eventos consecutivos: s, $s + 1$ y $s + 2$, $s \geq 1$, en los tiempos: t_s, t_{s+1} y t_{s+2}, iniciando la atención del paquete[3] en t_s, donde obviamente $a_i^k \leq t_s$. En t_s el paquete que estaba e la cola *i*, en el turno anterior a p_i^k, estaría saliendo de la cola (dicho paquete podría haber sido p_i^{k-1} en caso de que éste no hubiese sido extraído de la cola en su espera a ser atendido por alguna razón de operación especial de la cola).

Considérese que el tiempo de atención a p_i^k cruza al momento que se llama $s + 1$, que consiste en el inicio o en la conclusión de la atención de un paquete en otra cola diferente a la *i*) que es t_{s+1}, y finalmente, la conclusión de la atención al paquete es el evento $s + 2$, en el tiempo t_{s+2}.

Supóngase que entre los tiempos t_s y t_{s+1} se atiende la fracción del paquete llamada:

$$Fracción_{[t_s, t_{s+1})}\left(L_i^k\right)$$

Y supóngase que entre los tiempos t_{s+1} y t_{s+2} se atiende la fracción final del paquete, llamada:

[3] Se considera que un paquete ha llegado cuando el mismo ha llegado completo, y que un paquete ha salido cuando el mismo ha concluido su proceso de salida, es decir, ha salido completo.

$$Fracción_{[t_{s+1},t_{s+2})}\left(L_i^k\right)$$

Dichas fracciones dadas en bits. Entonces, se cumple que

$$L_i^k = Fracción_{[t_s,t_{s+1})}\left(L_i^k\right) + Fracción_{[t_{s+1},t_{s+2})}\left(L_i^k\right)$$

Con esto, las fórmulas siguientes son inmediatas:

$$t_{s+1} = t_s + \frac{Fracción_{[t_s,t_{s+1})}\left(L_i^k\right)}{R\phi_i \Big/ \sum_{\phi_j \in B[t_s,t_{s+1})} \phi_j} \qquad \textbf{(4)}$$

La anterior ecuación es equivalente a:

$$\frac{t_{s+1}-t_s}{\sum_{\phi_j \in B[t_s,t_{s+1})} \phi_j} = \frac{Fracción_{[t_s,t_{s+1})}\left(L_i^k\right)}{R\phi_i} \qquad \textbf{(5)}$$

Y similarmente se puede obtener la ecuación siguiente:

$$\frac{t_{s+1}-t_{s+1}}{\sum_{\phi_j \in B[t_{s+1},t_{s+2})} \phi_j} = \frac{Fracción_{[t_{s+1},t_{s+2})}\left(L_i^k\right)}{R\phi_i} \qquad \textbf{(6)}$$

Combinando las fórmulas se obtiene:

$$\frac{t_{s+1}-t_s}{\sum_{\phi_j \in B[t_s,t_{s+1})} \phi_j} + \frac{t_{s+2}-t_{s+1}}{\sum_{\phi_j \in B[t_{s+1},t_{s+2})} \phi_j} = \frac{Fracción_{[t_s,t_{s+1})}\left(L_i^k\right) + Fracción_{[t_{s+1},t_{s+2})}\left(L_i^k\right)}{R\phi_i} = \frac{L_i^k}{R\phi_i} \qquad \textbf{(7)}$$

La anterior expresión se puede escribir como:

$$V\left(t_{s+2}-t_{s+1}\right)+V\left(t_{s+1}-t_{s}\right)=\frac{L_i^k}{R\phi_i} \qquad \textbf{(8)}$$

Donde $\frac{t_{s+2}-t_{s+1}}{\sum_{\phi_j \in B[t_{s+1},t_{s+2})} \phi_j}$ se representa con $V\left(t_{s+2}-t_{s+1}\right)$, y $\frac{t_{s+1}-t_{s}}{\sum_{\phi_j \in B[t_{s},t_{s+1})} \phi_j}$ se representa con $V\left(t_{s+1}-t_{s}\right)$.

$V(\tau)$ se define como el lapso de **tiempo virtual** correspondiente al lapso de tiempo real τ. Es importante recalcar que durante cada lapso de tiempo virtual el conjunto de colas ocupadas debe permanecer constante.

El tiempo virtual es un concepto que ayuda a formular matemáticamente el comportamiento del sistema GPS. En el sistema el tiempo real va avanzando y el tiempo virtual también, pero el avance del tiempo virtual depende de las colas que vayan estando ocupadas en los diversos lapsos de tiempo virtual.

Se observa que los dos lapsos de tiempo virtuales abarcan toda la duración de atención del paquete. Se observa también que la suma de dichos dos lapsos de tiempo virtuales es igual a $L_i^k / R\phi_i$ que es una duración virtual completa de la atención al paquete (desde que se empieza a atender hasta que se acaba de atender).

Curiosamente, resulta que esta duración virtual de atención al paquete se puede calcular sin requerir saber qué colas estarán ocupadas o vacías durante la atención del paquete. Esto es como decir que desde la llegada del paquete se puede calcular la duración virtual de su atención. Meditando un poco, desde luego que sería importante ir llevando la cuenta, cada vez que llega o se va un paquete en el sistema de colas, del paso o avance del tiempo virtual en dicho sistema.

Se establece entonces que:

$$\textit{Duración virtual de atención al paquete } i = L_i^k / R\phi_i \qquad \textbf{(9)}$$

Asimismo, con las ecuaciones anteriores se puede obtener una expresión para poder llevar la cuenta del avance del tiempo virtual.

Considérese que se conoce el tiempo virtual específico $V(t_{s-1})$ (aun no se ha dicho cómo se conoce), entonces el avance e tiempo virtual hasta justo antes del siguiente tiempo virtual $V(t_s)$ sería[4]:

$$V(t_{s-1}+\tau)=V(t_{s-1})+\frac{\tau}{\sum_{\phi_j\in B[t_{s-1},t_{s-1}+\tau)}\phi_j}\quad,\quad 0<\tau<t_s-t_{s-1},\ s=2,3,... \tag{10}$$

Si el tiempo virtual del primer evento $t_1 = 0$, entonces se define $V(t_1) = V(0) = 0$.

Todos los tiempos virtuales subsecuentes correspondientes a los eventos se pueden calcular. Empezando con $V(t_2)$.

La ecuación (10), justo cuando τ alcanza el valor $t_s - t_{s-1}$ ofrece el valor de $V(t_s)$:

$$V(t_s)=V(t_{s-1})+\frac{t_s-t_{s-1}}{\sum_{\phi_j\in B[t_{s-1},t_s)}\phi_j},\ s=2,3...,\quad V(t_1)=0. \tag{11}$$

Es justo la ecuación (11) la que indica cómo ir calculando el tiempo virtual. Es suficiente actualizar el tiempo virtual cada vez que sucede un evento y no continuamente conforme pasa el tiempo real. Esto es porque la ocurrencia de los eventos es lo que puede cambiar el conjunto *B*, y solo dicho cambio altera el ritmo del tiempo virtual: un lapso de tiempo real en donde hay más sesiones (colas) con trabajo pendiente se convierte en un lapso de tiempo virtual mayor (que transcurre más lentamente).

Si los tiempos virtuales de inicio de atención y de finalización de atención de un paquete que llegó en el tiempo real a_i^k se denotan con S_i^k y F_i^k, respectivamente, de las formulaciones anteriores se puede indicar:

$$S_i^k=\max\left(F_i^{k-1},V\left(a_i^k\right)\right) \tag{12}$$

y

$$F_i^k=S_i^k+\frac{L_i^k}{R\phi_i} \tag{13}$$

[4] Ecuación (10) de [1].

Se puede establecer la siguiente regla. Al llegar p_i^k a la cola *i*, si se conoce ya el tiempo virtual F_i^{k-1}, se puede calcular en ese momento los valores S_i^k y F_i^k.

Considérese que el sistema GPS está un tiempo real *t*, correspondiente a un virtual tiempo virtual *V*(*t*), y que, $t_{s-1} \le t \le t_s$, es el tiempo real del último evento que ha ocurrido. La ecuación (11) sigue siendo válida usando *t* en lugar de t_{s-1}, obteniendo con ello:

$$V(t_s) = V(t) + \frac{t_s - t}{\sum_{\phi_j \in B[t, t_s)} \phi_j}$$

Considérese que *t* es el tiempo presente y lo que vaya a pasar adelante de *t* no se conoce. En ese momento se están atendiendo varias colas, y hasta el momento *t* el evento en t_s corresponde necesariamente a un evento de salida de paquete más próximo, porque no se sabe si va a llegar uno o más paquetes antes del tiempo t_s. Si entre *t* y t_s hubiese una llegada de paquete, el tiempo de ese evento se debería llamar ahora t_s, y el tiempo de salida del siguiente paquete a salir se debería recalcular porque el conjunto *B* posiblemente cambiara con la llegada del paquete, y además porque ese paquete recién llegado podría ahora ser el paquete a salir.

De cualquier forma, en el tiempo *t*, dado que no se conoce qué va a pasar antes del tiempo t_s, al tiempo t_s, llámesele F_{min}, y a t_s, para mayor claridad de nomenclatura, llámesele *Next*(*t*).

Entonces, la ecuación (11) puede escribirse como:

$$F_{\min} = V(t) + \frac{Next(t) - t}{\sum_{\phi_j \in B[t, Next(t))} \phi_j} \qquad \textbf{(14)}$$

Lo que nos permite despejar *Next*(*t*), que es el tiempo real del siguiente evento de salida de paquete.

$$Next(t) = t + \left(F_{\min} - V(t)\right) \sum_{\phi_j \in B[t, Next(t))} \phi_j \qquad \textbf{(15)}$$

En la operación de un simulador, la lista de programación de eventos "Scheduler" indica los tiempos reales en donde ocurren eventos, entre éstos aquellos de llegada de paquetes a cola, y eventos de salida de paquetes de colas (finalización de atención de los mismos)

Si en el tiempo real *t* ha ocurrido un evento de llegada o de salida, con relación a alguna cola cualquiera, de un paquete, en ese momento se puede calcular y conocer el tiempo del evento de la próxima salida de cola de un paquete, el cual se puede guardar en el Scheduler, el cual es justamente *Next*(*t*). Si entre el tiempo *t* y el tiempo *Next*(*t*) sucediese algún evento

de llegada a alguna cola, de un paquete, entonces se tendría que actualizar el tiempo *Next*(*t*), dado que el próximo evento de salida de paquete podría cambiar.

En el caso en que el proceso de simulación llegase al tiempo *Next*(*t*) en el Scheduler antes de la llegada de un nuevo paquete a cola, entonces el proceso de atención mismo del Scheduler quitaría ese tiempo del Scheduler y solamente se tendría que calcular el tiempo *Next*(*t*) del siguiente evento de salida de un paquete, para ponerlo en el Scheduler.

Sobre el procedimiento planteado, se hacen las siguientes observaciones:

- Durante cualquier lapso con trabajo en el despachador, el tiempo virtual es una función creciente estrictamente del tiempo real.

- Si se realiza el procedimiento anterior se podrán marcar los paquetes en orden creciente del tiempo virtual de su salida de cola (de la finalización de su atención), justo en el momento en que éstos llegan a cola.

- Mientras mayor peso tenga la cola en donde llega un paquete menor será el tiempo de espera del mismo.

PGPS es una forma de despacho que conserva la energía, en donde se atiende a un paquete a la vez con una tasa *R* (tasa total del enlace), y se sigue la pauta que indique un despachador GPS asociado que maneja la misma tasa total *R* del despachador PGPS, y el mismo tráfico de entrada al despachador PGPS. Dicha pauta es la lista ordenada de los tiempos virtuales en donde los paquetes deberían haber acabado de ser atendidos. Si el despachador PGPS operase en un dispositivo de red "en producción", dentro del mismo dispositivo debería correr una versión simulada de GPS que diese la pauta mencionada. Cuando el despachador PGPS, en el enrutador, tuviese que seleccionar al próximo paquete a atender, seleccionará a aquél que esté marcado con el siguiente menor tiempo de conclusión de atención, según el despachador GPS.

El despachador PGPS no conoce de pesos sino depende de la lista de tiempos virtuales del sistema paralelo GPS. La información de la composición de pesos en las colas se encuentra incorporada en los tiempos virtuales asignados a los paquetes, en la lista de tiempos virtuales del despachador GPS.

Los tiempos entre actividad e inactividad total, entre un despachador GPS y su despachador PGPS derivado son siempre iguales. La relación entre estos tiempos está estudiada con detalle en [1].

La operación del despachador PGPS es difícilmente viable para operación "en producción", y se usa en versiones de simulación.

2 Despachador GPS con Pesos que Cambian según la Longitud de la Cola.

El ejemplo del estado de las colas del despachador GPS de la Ilustración 1 servirá de apoyo para la comprensión del procedimiento que sigue.

Debe quedar claro que aquí no se proponen los valores de los pesos nuevos. Para ello el lector deberá referirse a [Rep-Tec-2-Equations-STP-Method].

En la Ilustración 1 se representan 5 colas con paquetes en proceso de ser atendidos y paquetes en espera para ser atendidos. En cada cola, 1 y 2, hay un paquete en proceso de atención y uno en espera para ser atendido. En la cola 3 hay un paquete en proceso de atención y dos en espera para ser atendidos. En cada cola 4 y 5 hay un paquete en proceso de atención y ninguno en espera para ser atendido.

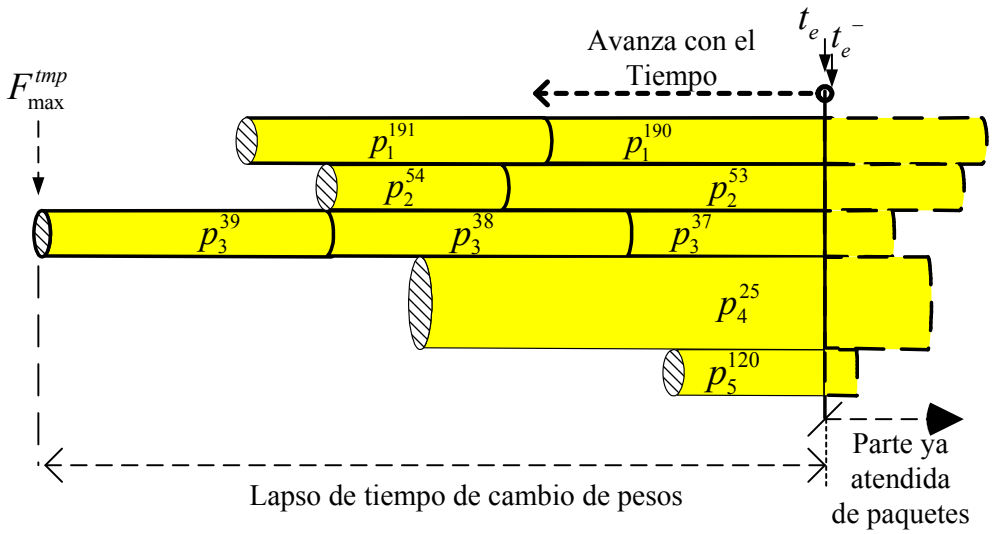


Ilustración 1

Se supone que se lleva el registro de la variable $F_{\max}$ como el tiempo virtual máximo de salida de un paquete, el cual se actualiza en el momento de la llegada o la salida de un paquete.

El siguiente algoritmo indica la forma de implantar el procedimiento de cambio de pesos con el despachador WFQ. Se incluyen en el algoritmo detalles de implantación en el código de C++ producido en [2] para el simulador ns-2. Todo el código del despachador WFQ indicado, incluyendo el algoritmo para modificar su operación y que opere con cambio de pesos, ha sido instalado en dicho simulador, por Alfredo Mateos.

3 Algoritmo Propuesto para Sistema de Despacho GPS con Pesos que Cambian Según la Longitud de las Colas.

3.1 Algoritmo con Detalles de Implantación.

Considérese que el número de colas es N[5].
Considérese que el número de colas es N. Durante la operación del algoritmo, se tienen dos estados: 1- NO hay requerimiento de cambio de pesos, 2- SI se debe cambiar de pesos en la siguiente oportunidad en que el cambio sea viable.

Inicialmente se declara lo siguiente[6]:

$F_{\max}^{tmp}$ es el tiempo máximo virtual de salida que se tiene de un paquete que se tenía registrado en el momento en que se cambió el estado de NO a SÍ se debe cambiar de pesos. Este tiempo virtual indica el tiempo en que se ha de cambiar de pesos desde el cambio de estado[7].

$F_{\max}$ es el tiempo máximo virtual de salida de un paquete que se actualiza en cada llegada o salida de paquete[8].

ϕ_j, $j = 1, ..., N$ son los pesos presentes[9].

ϕ_j^{New}, $j = 1, ..., N$ son los pesos nuevos[10] que va a sustituir a los pesos presentes. Existen también unos pesos iniciales ϕ_j', $j = 1, ..., N$, que son fijos[11].

---

[5] El número de colas máximo que se puede manejar en ns-2 está indicado en la variable fija MAX_QUEUES, que se designa igual a 8 en el archivo dsred.h. Posteriormente se declara la variable numQueues_, cuyo valor puede cambiarse desde OTcl y que debe ser menor o igual a MAX_QUEUES. El valor N que se utilizará será el de numQueues_, y las colas estarán numeradas de 0 a numQueues_ - 1. En [3] Pág. 9 y Pág. 34 se explica cómo cambiar dicho valor. Este valor se cambia en un objeto después de creado el mismo, pero, como se dice en la página 26 del manual de ns-2 [4], ns-2 garantiza que los valores actuales de las variables, tanto en los objetos interpretados como en los compilados, mantengan valores idénticos en todo tiempo. Ver también, en la misma referencia, las páginas 24, 25 y 26 sobre inicialización de las variables compiladas.

[6] Declaracón en la codificación con inserción clave “a”, en el archivo dsred.h, en la definición de la clase dsREDQueue. Los valores iniciales de estas variables se dan en el constructor de dsREDQueue en la inserción clave “b” en el archivo dsred.cc. La definición de estas rutinas está en el archivo dsred.cc en la inserción clave “m”.

[7] La variable a utilizar en el programa es: double Fmaxtmp.

[8] La variable a utilizar en el programa es double Fmax.

[9] Las variables a utilizar en el programa son: double queueWeightpr[MAX_QUEUES] (donde MAX_QUEUES es una variable preexistente que indica el número máximo de colas).

[10] Las variables a utilizar en el programa son: double queueWeightnw[MAX_QUEUES] .

[11] Los pesos fijos iniciales ya existían y eran int queueWeight[MAX_QUEUE] (declarados como protected en la clase dsREDQueue, en el archivo dsred.h).

wchpr ("Weight-change Process") es el indicador para saber si se está o no en un proceso o stado de cambio de pesos[12].

Se declaran rutinas varias para calcular las longitudes de las colas, los nuevos pesos, y para evaluar el criterio sobre el cambio de pesos.

3.1.1 Caso de la Llegada de un Paquete. Sistema de Despacho GPS.

Nota. Si se está en un proceso de cambio de pesos la llegada de un paquete no puede cancelar dicho estado.

CONDICION DE ESPERA INICIAL	0	Si el sistema esta ocioso, de forma inicial (antes de cualquier llegada de paquete) los valores de los pesos: ϕ_j , j = 1, ..., N, y de los nuevos pesos, ϕ_j^{New} , j = 1, ..., N, toman y se mantienen con el valor de los pesos iniciales ϕ'_j , j = 1, ..., N, pero si éstos últimos no sumasen 1, entonces los mismos pasarían su valor proporcionalmente ajustado para sumar 1. Tanto $F_{\max}^{tmp}$ y $F_{\max}$ toman el valor de cero[13]. Cuando llegue un paquete se va al paso A.

INICIO.	A[14]	En un tiempo dado $t = t_e$, se da un evento de llegada de paquete (controlado por el Scheduler)[15]. Se va al paso B.

Viene de A	B[16]	Parte existente en el algoritmo WFQ original. Se va a pasos C o D, dependiendo si el sistema estaba ocioso o no.

[12] Si la variable wchpr = 1 se está en proceso de cambio, si wchpr = 0 no se está en proceso de cambio. En la actual implantación en ns-2, el evento que ocurre justo después de cada segundo transcurrido de la simulación marca el momento en que se cambia el estado de NO a SI dentro del proceso de cambio de pesos. Después de que se hace el cambio de pesos dicho estado permanece en NO hasta el evento posterior al siguiente segundo transcurrido. Inicialmente existían otras formas de indicar cuándo se debería entrar en un estado de SI cambio de pesos. La forma actual utilizada se obtuvo con la guía de resultados de pruebas y evaluaciones, en simulaciones.
Las longitudes de las colas se evalúan con base de rutinas de promediado de colas. Los nuevos pesos se obtienen con el algoritmo de cambio de pesos presentado en [Rep-Tec-2-Equations-STP-Method].
En ns-2, desde el ambiente OTcl, se puede tener la opción para que ns-2 opere con cambio de pesos o sin cambio de pesos (ver la inserción "o" en el archivo dsred.cc). Por ejemplo, si el objeto (la cola ds tipo dsREDQueue) tiene el nombre, en OTcl, de "dsredq", con la instrucción: **$dsredq setChgWghtOpt 1** se puede cambiar la opción para que ns-2 opere con cambios de pesos.

[13] Estos valores iniciales se definen al crearse la cola (ver inserción clave "b" en el constructor de dsREDQueue e inserción clave "g" en la rutina reset de la clase dsREDQueue en el archivo dsred.cc), y NO se reinician cuando se llega a una situación de ociosidad (ver inserción clave "z" en WFQdequeueGPS de dsREDQueue en archivo dsred.cc).

[14] Esta parte del algoritmo ya existía en WFQenqueue(Packet *p, int queueid) de dsREDQueue (ver dsred.cc).

[15] Los procesimientos de llegada de paquete se ven en rutinas enque(Packet* pkt) y WFQenqueue(Packet *p, int queueid) ambas de dsREDQueue (en el archivo dsred.cc).

[16] Se usa la variable GPS_idle existente. GPS_idle = 1 quiere decir que el sistema sí estaba ocioso (ver WFQenqueue(Packet *P, int queueid) de dsREDQueue en el archivo dsred.cc).

Viene de interrogante B.	C[17]	Parte existente en el algoritmo WFQ original. Caso de que el sistema SÍ estaba ocioso. Se asigna el valor del tiempo virtual presente $V(t_e) = 0$. Se marca el sistema como no ocioso. Se va al paso E.

Viene de interrogante B.	D	Parte existente en el algoritmo WFQ original. Caso en que el sistema NO estaba ocioso (la siguiente formulación funciona para cuando SÍ o cuando NO se está en un proceso de cambio de pesos). Se calcula (según la ecuación (11) sustituyendo a t_s por t_e y a t_{s-1} por t_{e-1} $$V(t_e) = V(t_{e-1}) + \frac{t_e - t_{e-1}}{\sum_{\forall \ell \ni Cola_\ell(t_{e-1}) \in B[t_{e-1}, t_e)} \phi_\ell}$$ donde $\frac{t_e - t_{e-1}}{\sum_{\forall \ell \ni Cola_\ell(t_{e-1}) \in B[t_{e-1}, t_e)} \phi_\ell}$ es el incremento de tiempo virtual. En caso de estar en un proceso de cambio de pesos, $V(t_e)$ debe ser **menor que** $F_{\max}^{tmp}$ dado que al estar en este estado el evento correspondiente al tiempo virtual $F_{\max}^{tmp}$ no ha sucedido. Si $V(t_e)$ **fuese mayor o igual a** $F_{\max}^{tmp}$ **habría un error** y se debería detener el avance del algoritmo[18]. Notar que $V(t_e)$ se calcula usando los pesos actuales. **Cuidado.** Puede ser que por casualidad $V(t_e) = F_{\max}^{tmp}$ y aunque nunca $V(t_e)$ deberá ser mayor a $F_{\max}^{tmp}$ (o habría un error), la computadora, por errores de "precisión" podría tomar a $V(t_e)$ como mayor y enviar un mensaje de error, parando el programa (indebidamente) Esto se debe considerar en la programación del algoritmo[19]. Se va al paso E.

[17] Se asignan otras variables para control del algoritmo. Sobre los valores iniciales ver pie de página 16.
[18] Ver inserción clave "t", en dsREDQueue::WFQenqueue, en el archivo dsred.cc.
[19] Efectivamente esto sucedió en la programación del algoritmo en el simulador ns, para lo cual se tuvieron que tomar medidas.

Viene de C y D	E	Parte existente en el algoritmo WFQ original. La siguiente formulación funciona cuando SÍ se está o cuando NO se está en un proceso de cambio de pesos. Se calculan los tiempos de inicio y de terminación del paquete (para efectos de explicación se considera que llega el paquete #*k* a la cola #*i*, de tamaño L_i^k bits) Se evalúan S_i^k y F_i^k (notar que F_i^{k-1} y $V(t_e)$ son igual a cero si el sistema estaba ocioso). Se asigna[20]: $$S_i^k = \max\left(F_i^{k-1}, V(t_e)\right)$$ $$F_i^k = S_i^k + \frac{L_i^k}{\phi_i R}$$ Para el caso de sí estar en un proceso de cambio de pesos evidentemente $F_i^{k-1} \leq F_{\max}^{tmp}$, pero habrá que ver todavía si F_i^k es mayor o menor a $F_{\max}^{tmp}$. Se va al paso F.

Viene de E	F	Parte nueva en el algoritmo[21]. ¿Se está o no en proceso de cambio de pesos? (si no se está en este proceso NO se va a cambiar el procedimiento existente). Se va al paso G ó I, dependiendo.

Viene de interrogante F	G	Parte nueva en el algoritmo. En caso de que SÍ se está en un proceso de cambio de pesos se pregunta: ¿Se cumple que $F_i^k > F_{\max}^{tmp}$? (de cumplirse entonces lo anterior se va a cambiar el procedimiento existente). Se va al paso H ó I, dependiendo.

[20] En la rutina dsREDQueue::WFQenqueue, en el archivo dsred.cc, la fórmula original que se tenía se modificar para emplear, en lugar de los pesos originales queueWeight[ ], los pesos presentes queueWeightpr[ ] Ver código de modificación "f".

[21] Ver inserción clave "k" en WFQenqueue, en el archivo dsred.cc, para las líneas nuevas en el paso actual y en los dos subsiguientes de este algoritmo.

Viene de interrogante G	H	Parte nueva en el algoritmo. En caso de que $F_i^k > F_{\max}^{tmp}$, entonces no se acepta el valor de F_i^k. Se debe recalcular. $$Fracción\left(L_i^k\right)_{\left[S_i^k, F_{\max}^{tmp}\right]} = \left(F_{\max}^{tmp} - S_i^k\right)\phi_i \; R$$ $$Fracción\left(L_i^k\right)_{\left[F_{\max}^{tmp}, F_i^k\right]} = L_i^k - Fracción\left(L_i^k\right)_{\left[S_i^k, F_{\max}^{tmp}\right]}$$ El nuevo peso de la cola *i* entre los tiempos virtuales $F_{\max}^{tmp}$ y F_i^k es ϕ_i^{New}. $$F_i^k = F_{\max}^{tmp} + \frac{Fracción\left(L_i^k\right)_{\left[F_{\max}^{tmp}, F_i^k\right]}}{\phi_i^{New} R}$$ Combinando, se obtiene la ecuación completa: $$F_i^k = F_{\max}^{tmp} + \frac{L_i^k - \left(F_{\max}^{tmp} - S_i^k\right)\phi_i R}{\phi_i^{New} R}.$$ Se va al paso I.

Viene de interrogante F, de interrogante G, y de H	I	Parte existente en el algoritmo WFQ original. Se actualiza el conjunto *B*, y la sumatoria $\sum_B \phi_j$ (siempre con los pesos actuales –no con los nuevos)[22]. El conjunto *B* tendrá validez a partir del evento de llegada que acaba de suceder. Se actualiza $F_{\max}$ (que necesariamente es mayor a $F_{\max}^{tmp}$). Se va al paso J.

Viene de I	J	Parte existente en el algoritmo WFQ[23]. Se obtiene $F_{\min}$ y se recalcula $Next(t)$[24]. En el caso de estar en un proceso de cambio de pesos, como $F_{\min} \leq F_{\max}^{tmp}$. Entonces es válido hacer este cálculo con los pesos actuales. $$Next(t) = t_e + \left(F_{\min} - V(t_e)\right) \sum_{j \in B\left[t_e, Next(t)\right]} \phi_j$$ Se va al paso K.

[22] Se pone la información del evento de llegada en las tablas GPS y PGPS. Se guarda el par (número de cola, F_j^k), ordenando según el tiempo virtual de finalización. Con estas tablas y con la rutina get_key_max insertada nueva, para obtener el tiempo de atención virtual máximo que se lleva, se puede obtener fácilmente el valor de $F_{\max}$ (ver inserción "c" en el archivo wfq-list.h).

[23] En WFQscheduleGPS en el archivo dsred.cc. $F_{\min}$ se obtiene con una consulta a la lista GPS.

[24] Se actualiza *Next*(*t*) en el Scheduler, recordando que dicho tiempo es único en su tipo en el Scheduler.

Viene de J	K	Parte nueva en el algoritmo[25] ¿Se está o no en proceso de cambio de pesos? Va a paso L o N, dependiendo.

Viene de interrogante K	L	Parte nueva en el algoritmo. En el caso de que NO estar en un proceso de cambio de pesos, se evalúa el criterio de cambio de pesos (ya con la nueva llegada del paquete). Se debe hacer la evaluación aun cuando el sistema estaba ocioso. ¿El criterio de cambio de pesos indica que se deben cambiar los pesos? Se va a paso M o N, dependiendo.

Viene de la interrogante L	M	Parte nueva en el algoritmo. En el caso de que el criterio de cambio de pesos indicase que SÍ se deben cambiar los pesos: o Se registra que se está en un proceso de cambio de pesos (para el próximo evento) o Se evalúan los nuevos pesos $\phi_j(t_e)$, los que se llaman ϕ_j^{New}, $j=1,...,N$ (a los pesos presentes se les sigue llamando ϕ_j, $j=1,...,N$) o Se asigna $F_{\max}^{tmp} = F_{\max}$. Se va a N.

Viene de interrogante K, de interrogante L, y de M.	N	Se queda en ESPERA DE SIGUIENTE EVENTO.

3.1.2 Caso de la Salida de un Paquete. Sistema de Despacho GPS.

INICIO.	A’	Parte existente en el algoritmo WFQ. En un tiempo dado t_e se da un evento de salida de paquete[26]. El sistema no puede haber estado ocioso. Se va a B’.

[25] Ver inserción “l” en WFQenqueue(Packet *p, int queueid) en el archivo dsred.cc, para las las líneas nuevas en el paso actual y en los dos subsiguientes en este algoritmo.

[26] El Scheduler llega a un evento de salida del despachador GPS. Existen otros eventos de salida de paquete para el despachador PGPS (manejo por paquete) en el Scheduler, que llevan un tiempo muy cercano pero no igual al del despachador GPS (manejo por fluido) El llevar las salidas en el despachador GPS sirve para actualizar los tiempos virtuales. El despachador PGPS no utiliza los tiempos virtuales de forma directa. Para los eventos relativos al despachador GPS el Scheduler corre la instrucción this->WFQdequeueGPS(e) de dsREDQueue, y para los eventos relativos al despachador PGPS el Scheduler corre la instrucción this-> deque( ) de dsREDQueue, que incluye la línea para correr la rutina selectQueueToDeque, que a su vez incluye la línea para correr la rutina WFQdequeuePGPS( ) Ver archivo dsred.cc (recordar que la razón para llevar un despachador GPS operando en

Viene de A’	B’	Parte existente en el algoritmo WFQ[27]. Se calcula $V(t_e)$ $$V(t_e) = V(t_{e-1}) + \frac{t_e - t_{e-1}}{\sum_{\forall \ell \ni Cola_\ell(t_{e-1}) \in B[t_{e-1}, t_e)} \phi_\ell}$$ El valor de $V(t_e)$ siempre se acepta, por lo siguiente. Si NO se está en proceso de cambio de pesos el valor de $V(t_e)$ se acepta sin problema. Si SÍ se está en proceso de cambio de pesos el valor de el valor de $V(t_e)$ no podría ser mayor a $F_{\max}^{tmp}$, pues de lo contrario, en algún evento anterior se debía haber detectado esto (caso en que $V(t_e)$ hubiese sido igual a $F_{\max}^{tmp}$), y se habría anulado el estado de SÍ estar en cambio de pesos. Por lo anterior, es válido usar los pesos presentes en la anterior ecuación (en este caso $V(t_e) \leq F_{\max}^{tmp}$). Se actualiza en la lista de eventos el tiempo real del próximo paquete a salir y su número de cola. Va a paso C’.

Viene de B’	C’	Parte existente en el algoritmo WFQ. Justamente en el tiempo real t_e se acaba de ir un paquete. Se actualiza entonces la suma de pesos, $\sum_{j \in B[t_e, t_{e+1})} \phi_j$, a partir de t_e hasta el siguiente evento, t_{e+1} (no se sabe aun el valor de t_{e+1}). ¿El sistema se torna ocioso? (se puede probar si $V(t_e) = F_{\max}$ o si $\sum_{j \in B[t_e, t_{e+1})} \phi_j = 0$ recordando que $B[t_e, t_{e+1})$ es el conjunto de colas que tienen paquetes, al menos uno, en el lapso de tiempo real $[t_e, t_{e+1})$. Se va a paso D’ o E’, dependiendo.

paralelo con el despachador PGPS es que con el despachador GPS se lleva exactamente la cuenta de los tiempos virtuales y de su relación con los tiempos reales).

[27] En WFQdequeueGPS(Event *e) de dsREDQueue. Se designan dichas líneas con la clave “d” (ver archivo dsred.cc). Event *e es un objeto del tipo de objetos ordenados por el Scheduler (ver archivo Scheduler.cc).

Viene de interrogante C’	D’	Parte existente en el algoritmo WFQ[28] y parte nueva[29]. En caso de que el sistema SÍ se torne ocioso (lo que corresponde a $V(t_e) = F_{\max}$ aunque así no se verifica), se inicializa todo, llevando los pesos actuales y los pesos nuevos a ser como los pesos fijos. Además se asigna $F_{\max}^{tmp} = 0$ y $F_{\max} = 0$, y se desactiva la indicación de que el sistema estaba en proceso de cambio de pesos. Todos los tiempos virtuales se hacen cero. Se concluye la rutina de salida de paquete en el sistema de despacho GPS. Se va a paso 0 CONDICION DE ESPERA INICIAL.

Viene de interrogante C’.	E’	Parte nueva en el algoritmo[30]. Caso de que el sistema NO se torne ocioso. Se pregunta si SÍ se está o no en proceso de cambio de pesos y si $V(t_e) = F_{\max}^{tmp}$. Esto es un paso muy crucial. Considerar que en el programa $V(t_e)$ puede ser muy cercano a $F_{\max}^{tmp}$, sin ser igual, pero la computadora podría considerar iguales ambas cantidades (eso depende de la “precisión” usada). Se va a paso F’ si se cumple que SÍ se está en un proceso de cambio de pesos y que $V(t_e) = F_{\max}^{tmp}$. Se va a G’ de otra forma.

[28] En WFQdequeueGPS(Event *e) (ver archivo dsred.cc)
[29] Ver inserción con la clave “z” en WFQdequeueGPS(Event *e) de dsREDQueue (ver archivo dsred.cc)
[30] Todos los pasos siguientes se ponen en WFQdequeueGPS(Event *e) Ver inserción “e” en archivo dsred.cc.

Viene de Interrogante E'.	F'	Parte nueva en el algoritmo. Caso de que SÍ se está en un proceso de cambio de pesos y que $V(t_e) = F_{\max}^{tmp}$, entonces se sabe que **se acabó** el lapso de cambio de pesos (lo que quiere decir que los pesos actuales o presentes ya se pueden sustituir con los nuevos pesos) así que: • Se actualizan los pesos actuales ϕ_j, $j = 1, ..., N$, con los pesos nuevos $\phi_j(t_e) = \phi_j^{New}$, $j = 1, ..., N$. • Se prende la indicación de NO estar en proceso de cambio de pesos. • No importa cómo quede $F_{\max}^{tmp}$. Se hace cero por orden. NO se toca $F_{\max}$. Se debe actualizar la suma de pesos que se tenía, $\sum_{j \in B[t_{e-1}, t_e)} \phi_j$, con la suma hecha ahora con los nuevos pesos que habrá a partir del tiempo real presente t_e. Se va a G'.

Viene de interrogante E' y de F'.	G'	Parte nueva en el algoritmo. Se pregunta si NO se está en un proceso de cambio de pesos. En este caso se tienen algunas opciones: 1- Se ha llegado de F' por lo que NO se está en un proceso de cambio de pesos. 2- Se ha llegado de E' y SÍ se está en un proceso de cambio de pesos pero $V(t_e) < F_{\max}^{tmp}$. 3- Se ha llegado de E' y NO se está en un proceso de cambio de pesos (no importa entonces los valores $V(t_e)$ y $F_{\max}^{tmp}$). En caso de SI estar en un proceso de cambio de pesos se va al paso I'. En caso de NO estar en un proceso de cambio de pesos se evalúa el criterio de cambio de pesos (ya considerando que salió el paquete en t_e). Se va al paso H' o I', dependiendo el resultado del criterio.

Viene de segunda interrogante G'.	H'	Parte nueva en el algoritmo. Si el criterio de cambio de pesos indica que SÍ se debe cambiar de pesos, entonces: • Se registra que SÍ se está en un proceso de cambio de pesos (para el próximo evento) • Se evalúan los nuevos pesos $\phi_i^{New} = \phi_i(t_e)$, $i = 1, ..., N$ (los nuevos pesos anteriores habían quedado ya resignados como los pesos presentes con los cuales se trabaja, en el paso F') • Se asigna $F_{\max}^{tmp} = F_{\max}$.

Viene de H'.	I'	Parte existente en el algoritmo WFQ[31]. Se obtiene el tiempo virtual de ocurrencia, $F_{\min}$, del evento futuro más próximo, y se recalcula el tiempo real correspondiente a dicho evento, $Next(t)$, el cual es igual al tiempo actual, t_e, más una adición o "delay": $$Next(t) = t_e + \left(F_{\min} - V(t_e)\right) \sum_{j \in B\left[t_e, Next(t)\right]} \phi_j$$ Se actualiza dicho tiempo en el Scheduler (recordando que dicho tiempo es único en su tipo en el Scheduler) **Fijarse** que el conjunto de pesos $B\left[t_e, Next(t)\right]$ pudo haberse actualizado si resultó, en la reciente salida de paquete, en el tiempo presente t_e, que $F_{\max}^{tmp} = V(t_e)$ Esto sucedió en el paso F'.

Viene de I'.	J'	Se queda en ESPERA DE SIGUIENTE EVENTO.

[31] En WFQscheduleGPS( ) (ver archivo dsred.cc modificado por Alfredo Mateos).

Apéndice 1. Operación de ns.

ns-2 es un simulador escrito en C++ y en el lenguaje traductor ampliado de Tcl, con funcionalidad orientada a objetos OTcl, esencialmente. Al escribir "ns" en la pantalla de la computadora (al correr ns) se crean muchas clases de C++ que representan piezas importantes para la operación de ns como paquetes y colas.

Estas clases quedan en memoria. Asimismo se genera una "instancia" de Otcl (escrita en C++) que se presenta como una interfaz interactiva al usuario, es decir, se instala OTcl como ambiente de interfaz para comunicación con el usuario. Se crean clases y rutinas en OTcl para comunicación con los objetos y rutinas de C++.

Normalmente, el usuario al escribir "ns" acompaña a dicha instrucción con el nombre de un archivo que es un "Script" de OTcl. Este Script es un archivo que contiene instrucciones en OTcl, especializadas para descripción de redes según las reglas de ns. Con estas instrucciones se describe la configuración de la red y se los parámetros relacionados con los elementos de la red. Este Script lo a leer e interpretar la instancia interactiva de OTcl.

Las instrucciones en OTcl pueden provocar que se generen objetos en el entorno de C++ a partir de las clases previamente establecidas en C++. Asimismo, se pueden generar objetos "compuestos" que se conforman meramente de objetos C++ interconectados (con apuntadores), por ejemplo un nodo.

La última instrucción del Script es la instrucción run que inicia la ejecución de un lazo "while". Este lazo sirve para ir repasando los eventos puestos en un Scheduler para atenderlos. El Scheduler es un objeto único en el simulador que mantiene una serie de elementos llamados eventos, que están ordenados. El orden se tiene según el tiempo "real" de atención marcado en cada elemento. Cada uno de estos eventos es un objeto también, y contiene algunos parámetros como el tiempo de atención, un identificador único, una referencia al siguiente y al anterior evento, y una referencia para un objeto tipo Handler.

Cada vez que se atiende un evento del Scheduler se ejecuta una rutina apuntada por el evento. Esta rutina se llama mediante una rutina llamada handle, la cual se ejecuta hasta su terminación. Una vez concluida la atención de un evento se regresa el control al lazo while para que se busque, en el Scheduler, el siguiente evento. Al atenderse un evento el tiempo actual del Scheduler toma el valor del tiempo marcado para ejecución de dicho evento (así va avanzando el tiempo del Scheduler).

La rutina ejecutadas por un evento puede poner uno o más eventos adicionales en el Scheduler, mediante la rutina Schedule, la cual toma como parámetros: 1- un valor de "delay" (siempre positivo) que es el tiempo que se agregará al tiempo actual del Scheduler para ponerlo como el tiempo de ejecución del evento, 2- una referencia a un objeto (tipo Handler) que contiene siempre una rutina llamada "Handle" la cual, a su vez, contiene la

rutina específica que se correrá al ejecutarse el evento (llámese a esta rutina Handle el manejador), 3- una referencia a un nuevo objeto tipo Event que es el evento en sí.

Cada vez que se pone (que se inserta) un evento en el Scheduler, este evento se ordena, mediante un procedimiento de funciones Hash, en el Scheduler, de tal forma que el Scheduler mantiene los eventos ordenados según el tiempo de ejecución marcado en los eventos.

Con la instrucción "AT", en el Script, se pueden poner en el Scheduler eventos específicos. Es necesario poner al menos un evento inicial en el Scheduler. Aquí se supone que dicho evento es la generación del primer paquete que va a generar la fuente. Este primer evento generará uno o más eventos subsecuentes en el Scheduler.

La última instrucción run del Script es la instrucción con la que se empieza a ejecutar el lazo while que va a ir repasando los eventos en el Scheduler.

Ejemplo de Operación.

Considere un ejemplo muy simple en donde hay solo una fuente de paquetes (fuente), un nodo y un receptor de paquetes (Sink). Una fuente genera paquetes y no tiene que manejar colas.

La cola de los nodos se maneja en los enlaces. En este caso, la cola del nodo para la salida a la ruta de interés se maneja en el enlace #2.

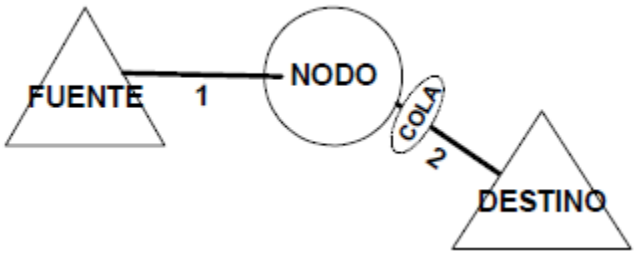


Ilustración 2. Una fuente, un nodo, un destino y dos enlaces.

Considere que el primer evento se ejecuta: la generación y despacho de un paquete por parte de la fuente. Se ejecutan los siguientes pasos.

1- Se genera un objeto tipo paquete (tipo Packet).

2- Se coloca en el Scheduler un evento de recepción de paquete[32] para hacer el encolamiento del paquete en la cola del enlace #2. Este evento está marcado para ejecutarse después de que haya pasado el tiempo de transmisión del paquete por el enlace #1 y del tiempo de propagación por el enlace #2.

[32] La puesta en el Scheduler se hace mediante las rutinas send y recv de la clase LinkDelay que se define en el archivo delay.cc [ns-2 Pg. 84, cap. 8]. El evento puesto en el Scheduler debe ser una rutina recv tipo Queue que contiene inicialmente la rutina enque.

3- Se coloca en el Scheduler un evento para desbloquear la cola "corriente arriba" (en este ejemplo no hay cola corriente arriba pero podría haberla en el enlace #1). Ese evento está marcado para ejecutarse después de que haya pasado el tiempo de transmisión del paquete pero antes del inicio de la etapa de propagación del paquete por el enlace #2[33]. Dicho evento habrá de correr la rutina resume de la clase Queue, la cual produce el desbloqueo, y antes del mismo produce también un deque de dicha cola corriente arriba para la liberación del siguiente paquete de esa cola (del enlace #1). Luego de todo esto, dicha rutina generaría otro evento para indicar la recepción del paquete en la cola del enlace #2[34].

4- Se coloca en el Scheduler el evento de generación y despacho del siguiente paquete que emitirá la fuente. El tiempo de ejecución de este evento será dependiendo de los algoritmos de generación de paquetes propios de la fuente y será un tiempo mayor al tiempo en que haya salido de la fuente el paquete que justamente se ha enviado.

Los siguientes eventos pueden ser cualquiera de los eventos recientemente generados.

Considérese que el siguiente evento es el evento de recepción de paquete en la cola del enlace #2[35], que implica un encolamiento del paquete en la cola del enlace #2 (con una rutina enque y otras derivadas que "anotan" la referencia del objeto del paquete en la cola).

Si la cola del enlace #2 se encuentra bloqueada al momento del encolamiento no se genera evento alguno[36]. Si la misma no está bloqueada al momento del encolamiento se realiza inmediatamente un inicio de desencolamiento, poniendo en el Scheduler un evento de recepción de paquete por el próximo elemento corriente abajo (podría ser otra cola). Este evento está marcado para ejecutarse después de que haya pasado el tiempo de transmisión del paquete por el enlace #2 y el tiempo de propagación por dicho enlace corriente abajo. Finalmente se bloquea la cola (para indicar que se está en estado de transmisión de este último paquete).

Notas sobre las Colas.

---

[33] Esto se hace con la última línea de la rutina recv de la clase LinkDelay en el archivo delay (que es: s.schedle(h, &intr_, txt);) donde txt es el tiempo de transmisión.

[34] Esto se hace mediante una rutina recv de LinkDelay que pone en el Scheduler un evento de recepción de paquete.

[35] Sin revisar a fondo entiendo que el manejador asociado a este evento es la función recv de la clase Queue. Se puede ver la declaración y definición de esta función en el archivo queue.h y queue.cc y en la página 70 del manual ns, capítulo 7.1.1.) Esta función primero hace un enque.

[36] En este caso deberá haber en el Scheduler un evento que indica cuándo se acabará de enviar el paquete que se está enviando. All llegar a dicho evento en el Scheduler se habrá de correr qh_.handle(event *) de clase Queue, que corre, a su vez: *this.resume, que a su vez corre Packet *p = deque( ) de clase Queue. Si de esta última corrida se recibe un objeto tipo paquete no nulo, entonces se corre recv(p, &qh_) del objeto (cola) que está corriente abajo. Este recv corre primeramente enque(p), que es una rutina virtual que será sustituida por la rutina correspondiente al objeto de la clase derivada (como podría ser dsREDQueue).

Los procedimientos anteriores ven las colas como objetos simples, sin embargo, dichas colas pueden estar compuestas. Por ejemplo, se puede tener un conjunto de colas DS (cola tipo dsREDQueue para ambientes DS) que en realidad es un conjunto de colas red (tipo redQueue) que estos procedimientos ven como una cola.

Entonces, cuando uno de estos procedimientos pone en cola o quita de cola, un paquete, "no sabe" lo que está pasando en los procesos que manejan las colas. Por ejemplo, si la cola es tipo DS (realmente un conjunto de colas), uno de los procedimientos para manejo de esta cola es un seleccionador de cola, que opera según el despachador de colas que se utilice.

Toda cola está ligada a un objeto. La clase dsREDQueue (que es un conjunto de colas simples DS), tiene como clase superior jerárquica a la clase Queue que les agrega varias rutinas, pocas, pero importantes. Las colas simples pueden ser de varios tipos[37].

[37] Toda cola simple, ya sea DropTail, redQueue o REDQueue, tiene como base a una cola física FIFO (tipo PacketQueue) Esta cola física es un objeto que contiene paquetes ordenados (cada paquete es un objeto) Cada paquete apunta al siguiente formándose una estructura.

Apéndice 2. Detalles sobre la Operación de ns con el despachador WFQ.

Operación.

Nos interesa especialmente analizar la operación de ns con DS (Servicios Diferenciados), y usando un despachador WFQ.

Las clases y sus dependencias se presentan en el diagrama de la **Ilustración 3**.

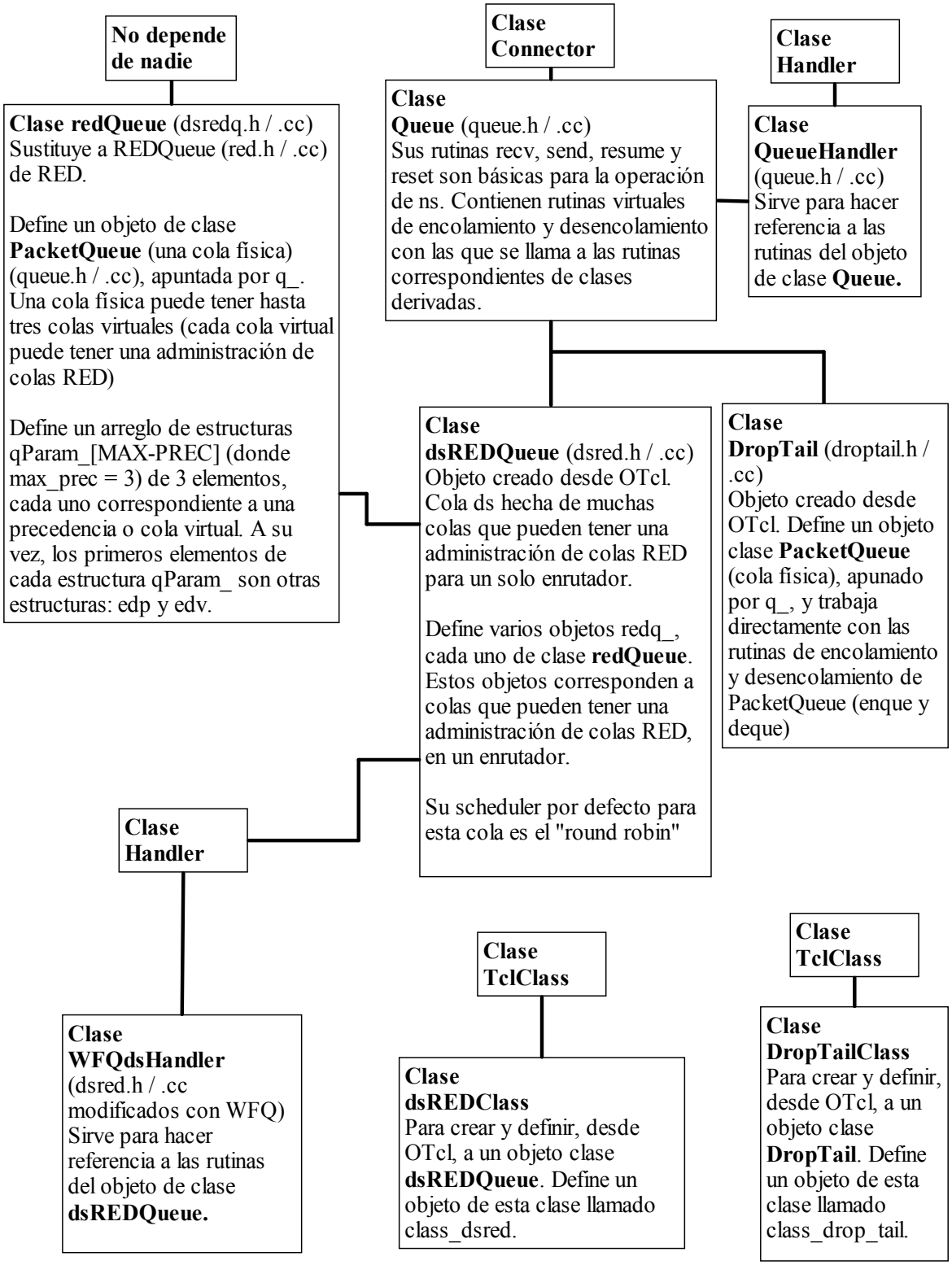
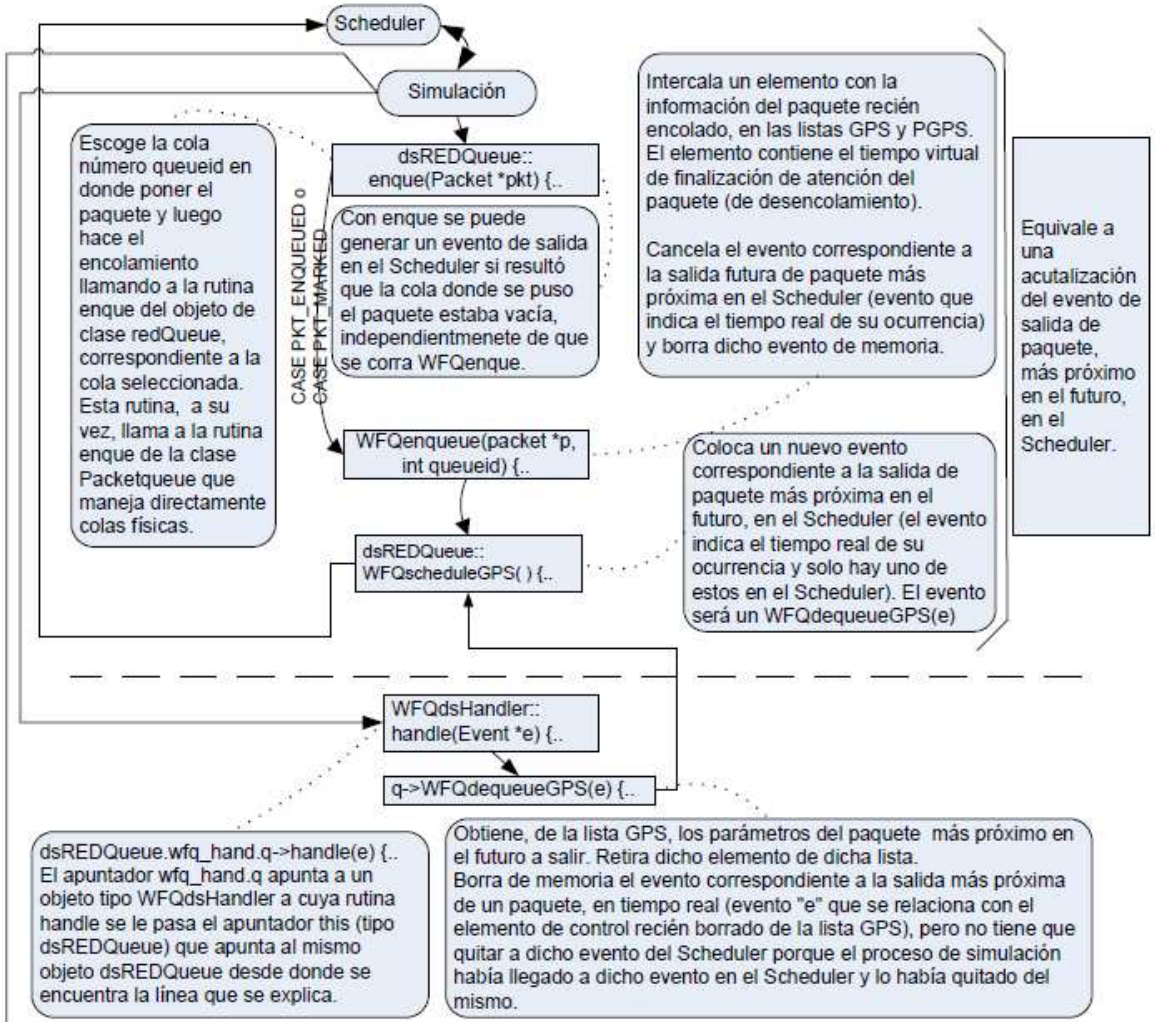


Ilustración 3. Diagrama de Dependencias de Clases de ns Relacionadas con la Operación de DS usando Despachador WFQ. (AMateos modifica esto para que se definan -en su lugar- varios apuntadores redqp_ a objetos de clase **redQueue**).

En la Ilustración 4 se presenta un diagrama de la operación de las rutinas de encolamiento y desencolamiento en ns operando con DS y despachador WFQ.



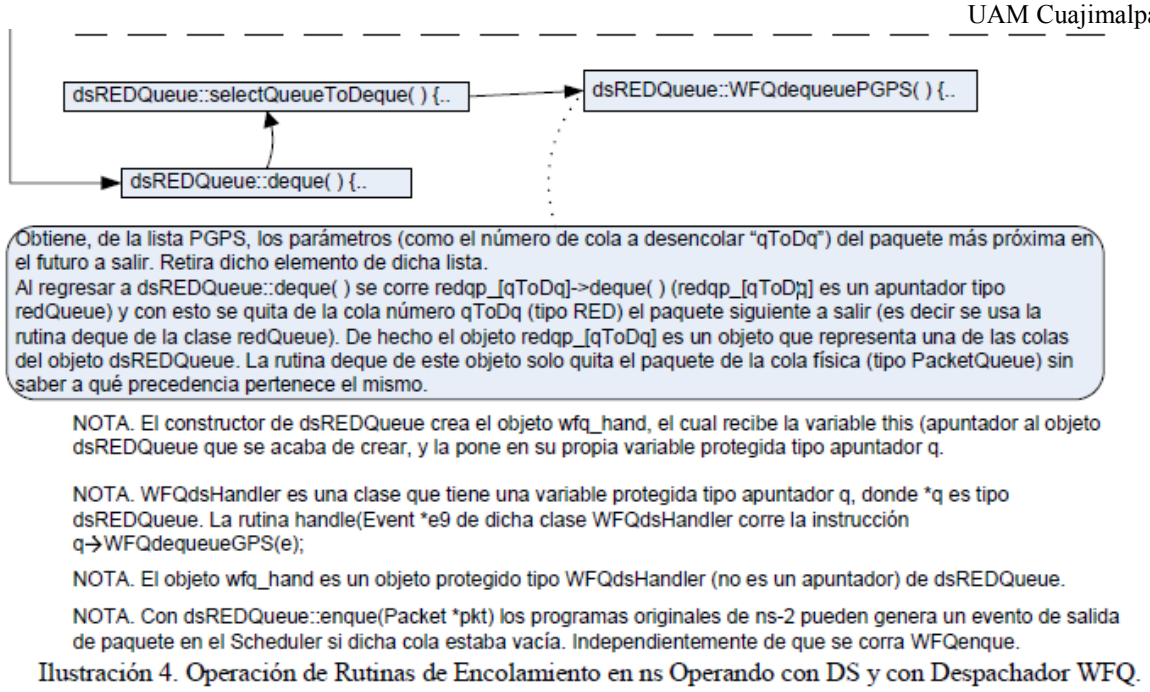


Ilustración 4. Operación de Rutinas de Encolamiento en ns Operando con DS y con Despachador WFQ.

Ilustración 4. Operación de Rutinas de Encolamiento en ns Operando con DS y con Despachador WFQ.

De la operación normal del simulador se van a ejecutar procesos de encolamiento y desencolamiento de manera continuada, llamando a las rutinas enque y deque de la clase Queue. Ambas rutinas en dicha clase son virtuales y serán sustituidas (preempted) por las rutinas enque y deque correspondientes al objeto de la clase derivada de Queue.

Si al hacerse un enque (de la clase Queue) en una cola resulta que ésta estaba vacía, se deberá hacer un deque de la cola y un enque en la siguiente cola "río abajo", considerando, para el deque, el tiempo que se tarde el paquete en acabar de salir considerando la tasa de transmisión, y para el enque el tiempo que se tarde el paquete en propagarse por el enlace de transmisión. Por esto, tanto el enque como el deque se deberán colocar en el Scheduler del simulador. Esto se hace con la rutina recv que existe tanto para la clase queue como para la clase linkdelay (en los archivos delay.h y delay.cc).

Al estar operando desde un objeto de clase dsREDQueue (objeto que define un conjunto de colas simples –donde cada cola es un objeto), cuando el simulador llama a las rutinas "enque" y "deque" de clase Queue, realmente se estarán llamando a las rutinas "enque" y "deque" de la clase dsREDQueue.

La operación para poner y retirar paquetes en una cola dsREDQueue (que es un objeto que a su vez contiene a varios objetos que representan cada uno a una cola simple), usando un despachador WFQ (GPS y PGPS), es como sigue:

Encolamiento de Paquetes.

Cuando el simulador encuentra en el Scheduler un evento de encolamiento, se ejecuta la rutina enque(Packet *pkt) de la clase Queue lo que genera una rutina enque(Packet *pkt) de la clase dsREDQueue. Lo anterior implica que se va a registrar la llegada de un paquete a una cola.

Entonces, dentro de dicha rutina primeramente se ejecuta otra rutina llamada enque(pkt, prec, ecn), a la que se convoca de la siguiente forma: redqp_[queue]->enque(pkt, prec, ecn))

Donde redqp_[queue] es una puntador a un objeto de la clase redQueue que representa a una cola, y donde queue es el número de cola a donde se va a poner el paquete. Dicho número de cola se obtuvo en la misma rutina enque de dsREDQueue mediante una búsqueda a una tabla PHB[38].

Posteriormente, en la rutina enque de la clase dsREDQueue se ejecuta la rutina WFQenqueue(pkt, queue) de dsREDQueue, en donde se calcula el tiempo virtual de encolamiento (y también el tiempo de desencolamiento dado que en el método WFQ se puede calcular este último tiempo desde el momento del encolamiento). Luego, información referente al paquete se intercala en las listas GPS y PGPS, especialmente creadas para la operación con WFQ, en un objeto que representa al paquete. Esta información es el número de cola en donde se puso el paquete y su tiempo virtual de desencolamiento. Dichas listas siempre están ordenadas por el menor tiempo virtual, es decir, el objeto a la cabeza tiene el menor tiempo virtual.

Al haber llegado un nuevo paquete a las colas del objeto dsREDQueue, el tiempo real del próximo paquete a salir en el sistema GPS debe actualizarse en el Scheduler del simulador, es decir, se cancela el evento que se tenía y se pone el nuevo evento relativo al desencolamiento del próximo paquete a salir en el sistema GPS, en el tiempo virtual para el sistema GPS (recordando que no debe haber simultáneamente más de un evento de este tipo en el Scheduler y que en el mismo el tiempo indicado está dado en tiempo real).

Como resumen, al poner un paquete en cola se calcula el tiempo virtual en que eso ocurre y se obtiene el tiempo virtual de desencolamiento. La información del paquete encolado, sobre su tiempo de desencolamiento y la cola donde se pone, se guarda en dos listas especiales de objetos: la lista GPS y la lista PGPS, objetos que están ordenados en ambas listas según el valor del tiempo virtual de desencolamiento (finalización de atención). Asimismo se actualiza en el Scheduler el evento de desencolamiento (finalización de atención) del paquete siguiente a desencolar en el sistema GPS.

Desencolamiento de Paquetes en el Sistema PGPS.

Cuando el simulador encuentra en el Scheduler un evento de desencolamiento para un paquete, lo que equivale a desencolar en el sistema PGPS, se ejecuta la rutina deque de la clase Queue, lo que genera se ejecute la rutina deque de la clase dsREDQueue. Esta última selecciona, mediante la rutina selectQueueToDeque de clase dsREDQueue, el número de cola (de objeto de clase redQueue) que se ha de tomar para desencolar. Para esto selectQueueToDeque corre la

[38] En la rutina original de WFQ [2] no menaje apuntadores pero esto causaba problemas de operación en ns por lo que Alfredo Mateos cambia las variables redq_[queue] a apuntadores redqp_[queue].

rutina WFQdequeuePGPS( ) con la que obtiene dicho número de cola a desencolar proveniente del objeto que encabeza la lista PGPS, aquél que contiene el menor tiempo virtual de la lista (tiempo de desencolamiento de un paqeute), eliminando dicho objeto de la lista. Por facilidad **se puede decir que la descarga, tanto de la cola como de la lista PGPS, se hace al mismo tiempo.**

Es importante observar que el desencolamiento mencionado no es para el sistema GPS (la lista GPS no se tocó). Este desencolamiento ha requerido del uso de la lista PGPS para conocer cuál es el próximo paquete a desencolar de entre todas las colas de clase redQueue.

El llevar una lista GPS sirve para poder calcular y actualizar el paso del tiempo virtual. Esto implica poner un evento único de desencolamiento con el menor próximo tiempo virtual en el Scheduler del simulador. En su proceso de revisión del Scheduler, la simulación encuentra este evento único y lo quita del Scheduler, corriendo, a su vez, la rutina WFQdequeueGPS(e) de dsREDQueue, la cual obtiene el objeto que encabeza la lista GPS que debe corresponder al paquete del evento retirado, borrando dicho objeto de la lista. De este objeto toma el número de cola en donde está el paquete y su tiempo virtual de desencolamiento -lo que sirve para actualizar el tiempo virtual). Asimismo, el simulador corre también la rutina WFQscheduleGPS( ) para poner un nuevo evento en el Scheduler correspondiente al desencolamiento del nuevo paquete con menor tiempo virtual para desencolar en el sistema GPS. En la lista GPS también queda ahora en la cabeza el objeto correspondiente a dicho paquete.

En suma, se debe notar que para el caso de un desencolamiento el tiempo virtual se actualiza cuando ocurre un desencolamiento en el sistema GPS y no en el sistema PGPS. Para actualizar en el Scheduler el evento del siguiente paquete a desencolar en el sistema GPS se debe haber pasado por un proceso de cálculo de tiempos virtuales que inicia con la obtención del tiempo virtual actual. Justamente para eso sirve llevar el control del sistema GPS.

RECAPITULACIONES.

LAS RUTINAS WFQdequeueGPS, WFQenqueueGPS no quitan ni ponen paquetes en colas. Estas rutinas interactúan con las listas de control GPS y PGPS y con el Scheduler del simulador. Dichas listas de control mantienen elementos que están ligados a los eventos de las salidas de paquetes en el sistema GPS. Cada elemento se relaciona con un paquete que está encolado, y contiene el tiempo virtual en donde el paquete debe salir del sistema, en el sistema GPS.

En el Scheduler hay, cuando mucho, un elemento del evento de la salida futura más próxima de un paquete, indicando el tiempo real de salida en la información del Scheduler. Cada vez que acontece un evento en la Simulación, ya sea de llegada o de salida de un paquete en el Scheduler, se debe actualizar el evento (único en el Scheduler) correspondiente a la próxima salida de un paquete en el sistema GPS.

La rutina dsREDQueue::WFQdequeueGPS no tiene que quitar al evento de próxima salida de paquete del Scheduler porque el proceso de simulación al llegar a dicho evento lo ha quitado,

pero sí Coloca un nuevo evento correspondiente a la salida de paquete más próxima en el futuro, en el Scheduler (indicando el tiempo real del evento).

La rutina dsREDQueue::WFQenqueue(packet *p, int queueid) {.. intercala un elemento de control nuevo, para el paquete recién encolado, en las listas GPS y PGPS. Dicho elemento indica el tiempo virtual de la finalización de atención del paquete.

En la Ilustración 5 se pone el diagrama de la operación de estas rutinas.

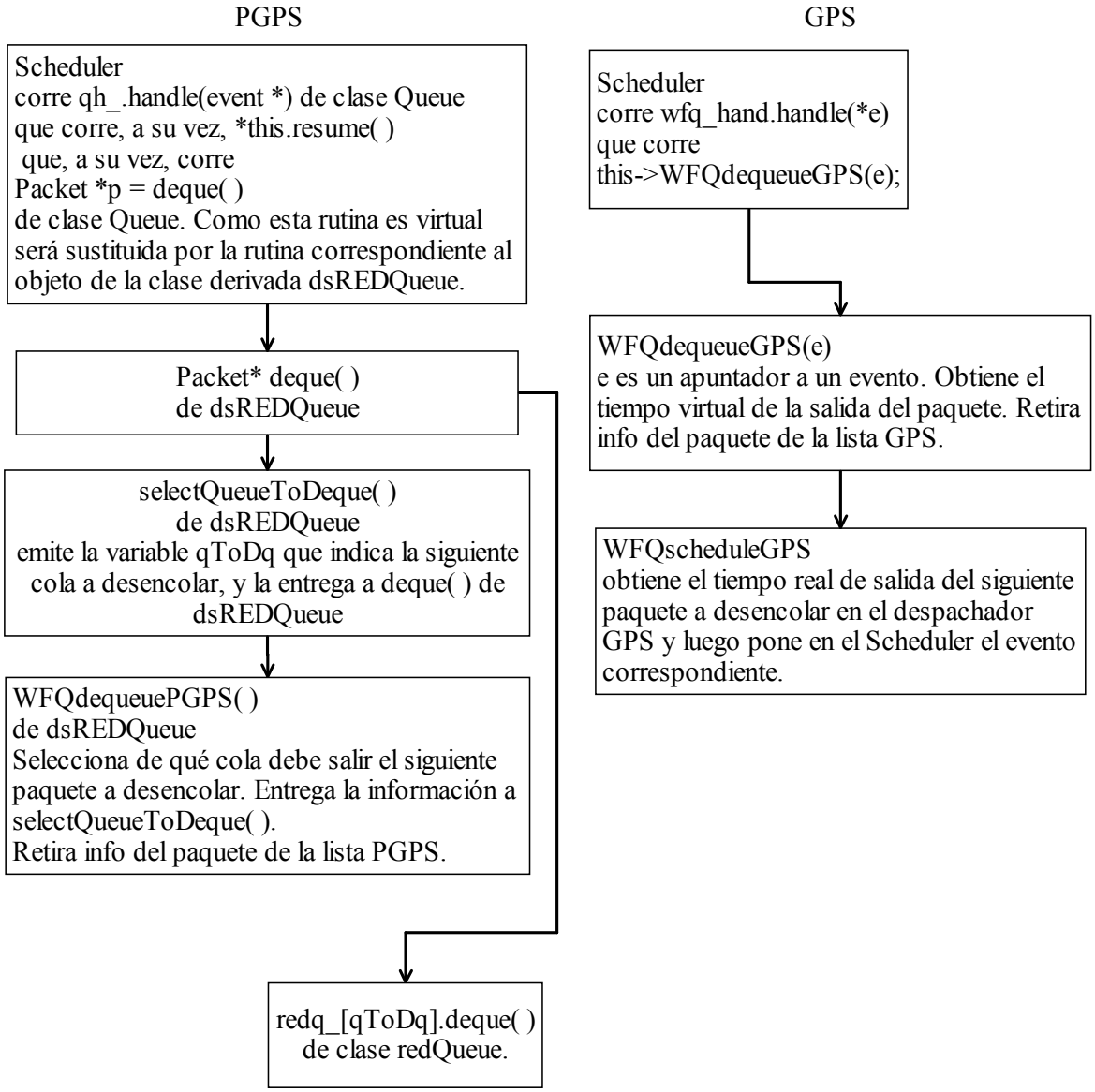


Ilustración 5. Desencolamiento con despachador WFQ. NOTA. El proceso redq_[qToDq].deque( ) de clase redQueue, simplemente quita el paquete de la cola física.

Apéndice 3. Detalles de Rutinas y Variables de ns Relativas a la Operación de Colas (antes de la modificación realizada por Alfredo Mateos)

Notas.

Cuando se escribe dsred.h modificado o dsred.cc modificado nos referimos a los archivos dsred.h y dsred.cc modificados, resultado del trabajo de MrKaic en 2001 [2] para operar originalmente con el algoritmo WFQ. Estos archivos fueron originalmente introducidos a ns por el grupo de trabajo de Nortel, Pieda et al [3] para que ns operase con DiffServ.

Clase: PacketQueue.
Clase Antecesora. TclObject.
Grupo de Archivos. De esta clase dependen todas las colase, iniciando por droptail.
Se Declara y Define en Archivo: queue.h / queue.cc
Clase Básica. Usa objetos tipo Packet (paquetes) que ya estaban definidos y usando espacio en memoria.

COLA FISICA FIFO. BASE DE TODAS LAS COLAS.

Esta es una clase que define una cola física FIFO. De esta clase depende cualquier tipo de cola más sofisticada que se tenga.

En la clase PacketQueue se tienen variables ("head_" y "tail_") que contienen las direcciones, inicial y final, respectivamente, de los paquetes de la cola. Los paquetes de la cola son objetos tipo Packet que están en memoria y que están en tránsito en el sistema. Cada paquete es un objeto que tiene una variable ("next_") la cual, a su vez, contiene la dirección del anterior paquete que está en cola (más próximo a ser atendido)

PacketQueue tiene algunas rutinas muy importantes como "remove" (que parece que quita un paquete específico de la cola al irlo buscando), y las rutinas virtuales "length" (regresa el valor de la variable "len_" que es la longitud de la cola física en paquetes), byteLength[39] (regresa el valor de la variable "bytes_" que es la cantidad de bytes que tiene la cola física), "enque" (pone a un paquete apuntado por un apuntador "p" en la cola física) y "deque" (quita de cola física el paquetes que correspondiente a la dirección indicada por la variable "head_" y regresa dicha dirección. (Ver queue.h)

Las rutinas enque y deque de PacketQueue aumentan y disminuyen, respectivamente, a la cola virtual FIFO, la contabilidad en paquetes (en la variable len_) y en bytes (en la variable bytes_)

---

[39] Para obtener los bytes del paquete se usa la rutina access de la clase hdr_cmn, y de dicha rutina se toma la variable size.

Hay otra rutina "remove" de PacketQueue, definida en queue.cc, que quita un paquete específico de la cola (va buscando)

Clase: Queue
Clase Antecesora: Connector
Grupo de Archivos: queue
Se Declara y Define en Archivo: queue.h / queue.cc
Define Objetos Tipo: PacketQueue.

CLASE QUE DEFINE RUTINAS ADICIONALES PARA MANEJAR UNA COLA FÍSICA.

Queue es una clase que define un objeto tipo PacketQueue (cola física) apuntado por el apuntador pq_. Las rutinas de Queue recv y send son las primeras rutinas que se usan para poner y quitar paquetes de colas generales. Estas rutinas usan, a su vez, rutinas virtuales queue y deque de la clase Queue que causarán se llame a las rutinas queue y deque de los objetos de clases derivadas desde los cuales se hagan las llamadas a las rutinas.

Clase: DropTail.
Clase Antecesora: Queue
Grupo de Archivos: droptail
Se Declara y Define en Archivo: droptail.h / droptail.cc
Define Objetos Tipo: PacketQueue. Usa rutinas de PacketQueue, de Queue.

Clase para definir una cola Droptail. Declara y define dos rutinas "enque" y "deque", que a su vez usan, las rutinas "enque" y "deque", consecutivamente, de la clase PacketQueue. Designa un objeto que es una cola física de tipo PacketQueue, apuntada por el apuntador q_.

Clase: dsREDQueue
Clase Antecesora: Queue
Grupo de Archivos: dsred
Se Declara y Define en Archivo: dsred.h / dsred.cc.
Define Objetos Tipo: redQueue.

Clase para definir un grupo de múltiples colas para operación con DS. Define muchas colas (muchos objetos) redq_[i], i = 1, MAX_QUEUES, de clase redQueue (ver dsred.h) Cada uno de estos objetos corresponde a una cola física. Los objetos (colas físicas) pueden haber sido declarados como colas dropTail, o como rio_c, rio_d, o wred. Esta declaración está inmersa en las rutinas de dsredq.h y dsredq.cc, donde se declara y define la clase redQueue.

En dsredq.h, al principio, se declara:
enum mredModeType {rio_c, rio_d, wred, dropTail};

Y en dsredq.h, dentro de la definición de la clase redQueue, se declara
mredModeType mredMode;

En dsredq.cc, se tiene la definición del constructor de la clase redQueue, en donde se tiene la instrucción:
mredMode = rio_c;

(todos estos modos tienen diferente forma de calcular su promedio de longitud de cola –ver dsredq.cc—)

En dsred.cc, se tiene la instrucción que inicia la definición de la rutina para cambiar el modo de la cola.
void setMREDMode(const char* mode)

En dsred.cc se tiene la instrucción para la interacción con Tcl:
if (strcmp(argv[1], "setMREDMode") == 0) {

Si mredMode es rio_c, rio_d o wred, entonces la cola física redq_[i] podrá tener varias colas virtuales.

Cada cola física está en espera de ser atendida, en un enrutador. Cada cola virtual de una cola física es una cola RED en varias versiones.

Esta clase tiene una rutina enque que a su vez llama a la rutina enque de la clase redQueue.

Esta rutina tiene la siguiente líneas de interés (se explica más sobre estas líneas en la página 39 y subsiguientes).

```
switch(redq_[eq_id].enque(pkt, prec, ecn)) {
  case PKT_ENQUEUED:
    break;
  case PKT_DROPPED:
    stats.drops_CP[codePt]++;
    stats.drops++;
    drop(pkt);
        break;
  case PKT_EDROPPED:
    stats.edrops_CP[codePt]++;
    stats.edrops++;
    edrop(pkt);
    break;
  case PKT_MARKED:
    hf->ce() = 1;          // mark Congestion Experienced bit
    break;
```

```
  default:
    break;
  }
}
```

Sobre las variables ecn y ecn_ (Explicit Congestion Notification)

La operación de ecn con relación a los resultados de la ejecución de la rutina de encolamiento: PKT_ENQUEUED, PKT_DROPPED, PKT_EDROPPED, PKT_MARKED se dan en la página 39 y subsiguientes.

La variable ecn_ se declara en la clase dsREDQueue (ver dsred.h) con la línea:
"int ecn_; // used for ECN (Explicit Congestion Notification)"

En dsred.cc existe un línea:
"bind_bool("ecn_", &ecn_);"
que lida a la variable ecn_ de OTcl que se liga con la variable ecn_ de C++.

También, en la rutina enque de dsREDQueue (Ver dsred.cc) se lleva una variable ecn que se relaciona con ecn_. Se puede ver esto en la línea:
" if (ecn_ && hf->ect()) ecn = 1;"

Luego se llama a la rutina enque del objeto redq_ de clase redQueue, pasándole el parámetro ecn, mediante la siguiente instrucción:
switch(redq_[eq_id].enque(pkt, prec, ecn)) { (ver dsred.cc)

Sobre el Despachador.

En dsred.cc se fija la variable schedMode = schedModeRR (Round Robin) Posiblemente dicho valor para la variable schedMode puede cambiar mediante la interacción con el usuario con Tcl.

En RR (schedMode = SchedModeRR) (Round Robin) se selecciona la cola siguiente de la que se había atendido antes, y si hay una cola vacía se sigue con la siguiente hasta encontrar una cola no vacía (parece que si todas las colas están vacías no se prevé una acción especial de esta rutina al respecto)

En WRR (schedMode = SchedModeWRR) (Weighted Round Robin) se envían tantos paquetes seguidos de cada cola como el peso que tenga la cola (llevado en queueWeight[qToDq] –donde qToDq es la cola que le toca ser atendida por el despachador) (se ve que el peso debe corresponder al número de paquetes seguidos que se quiere que se despachen y debe ser un entero) Si una cola está vacía la atención se va hacia la siguiente cola. La rutina no da prevención si todas las colas están vacías.

Hay otros despachadores como schedMode = schedModeWIRR y schedMode = schedModePRI (Priority Queueing) que no me interesan por el momento. (Ver todo esto en dsred.cc)

En dsred.cc hay una rutina que inicia con la siguiente línea:
void dsREDQueue::setSchedularMode(const char* schedtype) {
para poner el tipo de modo de despachador. En dsred.cc se tiene la instrucción que inicia el "if" para la relación con Tcl para obtener el valor del tipo de despachador:
if (strcmp(argv[1], "setSchedularMode") == 0) {

Hay una rutina dsREDQueue::addQueueWeights (int queueNum, int weight) (en dsred.cc) Se utiliza una ejecución de esta rutina para cada valor de peso (un valor de peso para cada cola) En dsred.cc se ven las instrucciones para la relación de esta rutina con Tcl. Se ve que los pesos entran directamente sin más procesamiento de parte de las rutinas.

Clase: REDQueue
Clase Antecesora: Queue
Grupo de Archivos: red
Se Declara y Define en Archivo: red.h / red.cc

Esta clase no se analiza aquí porque para DS es sustituida completamente por la clase redQueue. Las estructuras iniciales definidas en red.h son tomadas para definir la clase redQueue en dsredq.h y dsredq.cc.

Clase: redQueue
Clase Antecesora: No depende de otra clase.
Grupo de Archivos: dsred
Se Declara y Define en Archivo: dsredq.h / dsredq.cc
Define Objetos Tipo: PacketQueue.

ESTA CLASE PROVEE ESPECIFICACIONES PARA UNA COLA FÍSICA QUE PUEDE MANEJAR HASTA TRES COLAS VIRTUALES TIPO RED. ESTA CLASE ESTÁ HECHA ESPECIALMENTE PARA UTILIZARSE CON DS, Y HACE LAS VECES (SUSTITUYE COMPLETAMENTE) A LA CLASE REDQueue DE RED.

Con un objeto tipo redQueue se declara, define y lleva una cola física (FIFO) que es un objeto tipo PacketQueue que es apuntado por el apuntador q_ (recordar que la clase PacketQueue define una cola física básica FIFO en la cual se basan todas las colas) (Ver dsredq.h)

En la cola física se manejarán tres colas virtuales (cada una asociada a una precedencia) Cada cola virtual es una cola RED. Se lleva el tamaño promedio de cada cola virtual para decidir sobre su operación RED. Se lleva el tamaño físico de la cola en total. Cuando un paquete se va a encolar, éste se encola siempre al final de la cola física, es decir, su ingreso a la cola es FIFO, y al ingresar el paquete a la cola física se actualiza el tamaño promedio

pesado de la cola virtual correspondiente y el tamaño actual no pesado de la cola física (ver dsredq.cc en la rutina enque de redQueue)

La clase redQueue tiene una estructura qParam[MAX_PREC] (donde MAX_PREC = 3) (o sea que la estructura tiene tres elementos donde cada elemento corresponde a una precedencia de la cola) Los primeros elementos de qParam[i], para i = 0, 1 y 2, son: edp, y edv, que son, ambos, a su vez, estructuras, y que se definen en la clase REDQueue (edp son los "early drop parameters" y "edv" son los "early drop variables") (Ver red.h al principio)

En la clase redQueue se definen las rutinas "enque", "deque". Enque utiliza la rutina enque de PacketQueue.

Esta rutina enque es llamada, a su vez, por la rutina enque de dsREDQueue (ver dsred.cc)

Con la rutina enque de redQueue el paquete que llega a la cola se encola si la longitud real de la cola FIFO no ha llegado a su límite, y si además se cumple una de los tres casos siguientes, exclusivos:

1- Se está en modo de administración de congestión de cola "dropTail" y lalongitud real de la cola es menor al parámetro minth (ver la variable correspondiente de dsredq.cc)
2- Se está en cualquier otro modo de administración de congestión y la variable ecn está prendida (posteriormente se podría marcar el paquete como desfavorecido según procedimientos de probabilidad)

3- Se está en cualquier otro modo de administración de congestión, la variable ecn está apagada, pero mediante los métodos RED se ha determinado que el paquete no va a ser marcado o tirado.

Una explicación general del anterior procedimiento es:

De dsredq.cc. Para los casos rioc, riod, wred, para la rutina redQueue::eque(Packet *pkt, int pret, int ect) { ...

Si ecn está encendido un paquete siempre se pone en cola (antes de revisar qué pasará con el tamaño de la cola) de otra forma no se hace.

Cuando un paquete es designado para ser tirado o degradado, por probabilidad (cuando el tamaño promedio de la cola estaba entre minth y maxth), la variable ecn hace la diferencia entre regresar el mensaje PKT_MARKED (cuando ecn estaba encendido) o PKT_EDROPPED (cuando ecn estaba apagado)

Cuando un paquete es designado para ser tirado o degradado, en el caso de que el tamaño promedio de la cola fuese mayor a maxth, la variable ecn hace la diferencia entre regresar el mensaje PKT_MARKED (cuando ecn estaba encendido) o PKT_DROPPED (cuando ecn estaba apagado)

En todo caso, PKT_MARKED se refiere a un paquete que ya fue puesto en cola pero que después fue marcado (degradado)

Los casos de mensajes PKT_EDROPPED o PKT_DROPPED son casos en donde los paquetes nunca fueron puestos en cola ni se pondrán. Dichos paquetes no se degradan ni se tiran porque nunca fueron puestos en cola.

En caso de que el tamaño de la cola haya sido menor a minth, el paquete se pone en cola cuando ecn estaba apagado (pero no se vuelve a poner en cola si ecn estaba encendido porque ya se había puesto en cola)

Para cualquier paquete que se haya puesto en cola, y que no haya sido designado para tirarse se regresará el mensaje PKT_ENQUEUED.

Entonces:

PKT_MARKED quiere decir que el paquete sí se encoló pero que se seleccionó de forma desfavorable.

PKT_EDROPPED quiere decir que el paquete no se encoló y se seleccionó de forma desfavorable probabilísticamente, por lo que ya no se encolará.

PKT_DROPPED quiere decir casi lo mismo que PCKT_EDROPPED, pero en este caso la selección se hizo de forma no probabilística (el tamaño promedio de la cola virtual sobrepasó thmax)

A continuación se da la operación con más detalle:

Cada cola virtual tiene una longitud real actual llevada por la variable “qParam_[prec].qlen”, donde “prec” es el número de precedencia asociado a dicha cola virtual (de 0, 1 ó 2) La variable qlen es la longitud de la cola real actual, en “paquetes” (no es la longitud pesada) (Ver. dsredq.h)

El tamaño real de la cola física FIFO, en paquetes, que se lleva en q_->length(), donde q_ es un apuntador a la cola física FIFO, que es un objeto PacketQueue.

Si el tamaño real de la cola física FIFO, en paquetes, que se lleva en q_->length(), es mayor o igual al tamaño máximo que se puede tener en paquetes de dicha cola física, lo que se lleva en la variable qlim (ver dsredq.h), entonces el paquete no se encola, no se hace nada, y la rutina termina regresando el valor PKT_DROPPED (PKT_DROPPED y las otras posibilidades de regreso de esta rutina enque de redQueue deben estar asociadas a valores pues la rutina está definida como para regresar enteros) SI ESTÁN ASOCIADAS. En dsred.h se define

```
#define PKT_MARKED 3
#define PKT_EDROPPED 2
#define PKT_ENQUEUED 1
#define PKT_DROPPED 0
```

Si el tamaño real de la cola física en paquetes es menor al máximo tolerado qlim entonces se hace lo siguiente:

Si el modo de administración de congestión de colas que se emplea es "dropTail" (llevado en la variable "mredMode") entonces se realiza la llamada a la rutina enque de PacketQueue (para poner el paquete en la cola física FIFO) que se realiza con la instrucción "q_->enque(pkt);", donde debe recordarse que q_ es un apuntador a la cola física FIFO y pkt es un apuntador a un objeto tipo Packet (al paquete) Al final, esta rutina concluye regresando el valor PKT_ENQUEUED.

Para los otros casos de administración de congestión de colas, como: "rio_c", "rio_d", o "wred", a continuación, se calcula la longitud promedio de las colas virtuales (en paquetes) Dicha longitud se ubica en la variable qParam_[prec].edv_.v_ave (dónde prec indica la precedencia o cola virtual que se lleva) Dicho cálculo promedio se realiza según el modo de administración de congestión de colas que se lleve. Para esto se utilizan los valores de límites mínimo y máximo que se han fijado para cada cola virtual, llevados en las variables qParam_[prec].edp_.th_min y qParam_[prec].edp_.th_max.

Una vez obtenida esta longitud promedio, con la metodología de RED se decide si el paquete se ha de tirar o no o marcar o no, de la siguiente forma.

Si ecn está prendida, entonces sí se llama a la rutina enque de Packetqueue (para poner el paquete en la cola física FIFO) con la misma instrucción que aquí se ha comentado, y sin importar el tipo de otro casos de encolamiento mredMode se haya tenido, pero en este caso se aumenta, en uno, el valor real de la cola virtual, en paquetes, en donde se ponga el paquete, que se lleva en la variable qParam_[prec].qlen (longitud en paquetes)

Si el tamaño promedio de la cola, en paquetes, está entre qParam_[prec].edp_.th_min y qParam_[prec].edp_.th_max (parámetros de RED), y si por probabilidad resulta que el paquete debe ser tirado, entonces, sí ecn estaba prendido la rutina concluye regresando el valor PKT_MARKED, y si ecn estaba apagado la rutina concluye regresando el valor PKT_EDROPPED. Si el tamaño promedio de la cola es mayor a qParam_[prec].edp_.th_max entonces, si ecn estaba prendido la rutina concluye regresando el valor PKT_MARKED, y si ecn estaba apagado la rutina concluye regresando el valor PKT_DROPPED.

Si el tamaño promedio de la cola, en paquetes, es menor a qParam_[prec].edp_th_min, entonces el paquete no se marca para tirar. En este caso, si ecn estaba prendido entonces la rutina concluye regresando el valor PKT_ENQUEUED (porque el paquete ya había sido encolado en la cola física y virtual correspondiente), y si ecn estaba apagado entonces de

cualquier forma se llama a la rutina enque de PacketQueue (para encolar el paquete en la cola física FIFO) y también se aumenta en uno el tamaño en paquetes de la cola virtual correspondiente, y regresándose, también, el valor de PKT_ENQUEUED.

Rutina calcAvg(prec, m+1)

La rutina calcAvg(prec, m+1) (ver dsredq.cc) no regresa valor pero pone el promedio de la longitud de la cola virtual, prec, en paquetes, en qParam_[i].edv_.v_ave, donde i puede ser el número de la cola virtual actual prec, que puede estar entre 0 a numPrec – 1 (numPrec es el número máximo de precedencias que se usa), dependiendo del tipo de atención de congestión que se tenga, rio_c, rio_d, o wred. Para rio_d y rio_c se actualiza el tamaño promedio de la cola virtual, en paquetes, usando los tamaños reales, en paquetes, de las colas virtuales qParam_[i].qlen (ver dsredq.h), y para wred se actualizan el tamaño promedio de las colas usando el tamaño total de la cola FÍSICA FIFO, en paquetes, llevado en q_->length() (Ver queue.h en donde la rutina lenght() regresa la variable len_ que es el tamaño de la cola FIFO en paquetes) Las colas que se actualizan son: para el caso de rio_c se hace para las cola i de 0 a prec, para rio_d se hace para la cola prec, y para wred se hace para i de 0 a numPrec – 1.

Para el cálculo de esta longitud promedio, en paquetes, se utiliza la propuesta del artículo de Sally Floyd, donde el peso para actualizar la longitud es "qParam_[prec].edp_.q_w" (Ver dsredq.cc), y la variable *m* corresponde a la variable *m* también del artículo de Sally Floyd (*m* se usa cuando la cola se ha quedado vacía) Dicho valor promedio se queda en el parámetro qParam_[prec].edv.v_ave (Ver red.h)

En dsredq.h se define qParam como una estructura, y en la clase redQueue (ver dsredq.h) se define un grupo de variables qParam_[MAX_PREC], que tienen la estructura de qParam. qParam_[prec].edv_.v_ave es el tamaño promedio de la cola virtual prec. Esta estructura d qParam tiene, a su vez, las estructuras edv (Early Drop Variables) y edp (Early Drop Parameters) que se pueden consultar en red.h)

En dsredq.cc se asigna mredMode como rio_c inicialmente.

Esisten algunas rutinas que nos dan los valores de longitudes de colas sin tener que acceder a las variables (Ver dsredq.cc), que son.

> int redQueue::getRealLenght (void) que regresa el valor de la longitud real física de la cola FIFO, en paquetes.

> double int redQueue::getWeigthtedLength() da el valor promedio de toda la cola física. Me parece que esta rutina funciona bien para mredMode = rio_c y mredMode = rio_d, pero tengo mis dudas que la rutina opere bien para mredMode = wred, y desde luego no es para mredMode = dropTail, en paquetes.

double redQueue::getWeightedLenght_v(int prec) que entrega la longitud promedio de la cola virtual prec, en paquetes.

int redQueue::getReal Lenght_v(int prec) que da el valor real de la longitud de la cola virtual, en paquetes.

Existen otras dos rutinas para fijar el "packet time constant" y el "mean packet size" que se debe tener en cuenta.

En dsredq.h, al principio, se declara:
```
enum mredModeType {rio_c, rio_d, wred, dropTail};
```

Y en dsredq.h, dentro de la definición de la clase redQueue, se declara
```
mredModeType mredMode;
```
(ver donde se explica dsREDQueue en este documento para más detalles)

Clase: REDQueue
Clase Antecesora:
Grupo de Archivos: red
Se Declara y Define en Archivo: red.h / red.cc

COLA RED. HECHA ANTES DE QUE SE INCORPORASE DIFFSERV EN NS.

Es sustituida completamente por redQueue cuando se está usando DiffServ.

Clase: dsREDClass.
Clase Antecesora: TclClass.
Grupo de Archivos: dsred
Se Declara y Define en Archivo: dsred.cc

Liga la nueva clase dsREDQueue con una clase del mismo nombre de Tcl. Contiene una rutina llamada create que crea en memoria espacio para un objeto llamado dsREDQueue en Tcl, y regresa, en C++, un apuntador a dicho objeto (el ancestro de dicho objeto es TclObject), por eso la llamada a dicha rutina es:

```
TclObject* create(int, const char*const*) {
  return (new dsREDQueue);
}
```

Y crea el objeto class_dsred tipo dsREDClass, en C++.

Clase: QueueHandler.
Clase Antecesora: Handler.
Grupo de Archivos: queue.
Se Declara y Define en Archivo: queue.h / queue.cc

*Cuando se crea un objeto tipo Queue se reserva en memoria espacio para sus variables (no para sus rutinas que solo están en memoria una vez) como parte de la estructura de la clase Queue. Una de las variables del objeto es el objeto protegido qh_ que es de tipo QueueHandler (ver queue.h) A continuación se ejecuta el constructor de Queue para dicho objeto (ver queue.cc) en donde qh_ recibe el valor *this (this es el apuntador al objeto Queue) por lo que qh_ recibe al mismo objeto tipo Queue. El constructor del objeto tipo Queue hace otras cosas como varios binds. Al crearse el objeto qh_ (tipo QueueHandler), a su vez, se reserva espacio en memoria para sus variables (similar a como se hizo con el objeto tipo Queue referido) y se ejecuta el constructor de QueueHandler para qh_, y finalmente la variable privada queue_, tipo Queue, del constructor de qh_, acaba referenciada al objeto tipo Queue creado inicialmente.*

Entonces, cuando se corre la rutina pública handle(Event*) de QueueHandler se corre la rutina pública resume de queue_ (ver queue.cc) con la instrucción única:
queue_.resume();

Dicha rutina tiene un argumento de entrada Event* pero no lo utiliza para nada.

Clase: WFQdsHandler
Clase Antecesora: Handler
Grupo de Archivos: dsREDQueue modificado
Se Declara y Define en Archivo: dsred.h modificado / dsred.cc modificado.

Cuando se crea un objeto tipo dsREDQueue se reserva en memoria espacio para sus variables (no para sus rutinas que solo están en memoria una vez) como parte de la estructura de la clase dsREDQueue. Una de las variables del objeto es el objeto protegido wfq_hand que es de tipo WFQdsHandler (ver dsred.h modificado) A continuación se ejecuta el constructor de dsREDQueue para dicho objeto (ver dsred.cc modificado) en donde el objeto wfq_hand recibe el valor this (this es el apuntador al objeto dsREDQueue mismo por lo que el constructor de wfq_hand recibirá un apuntador al objeto dsREDQueue creado) El constructor de dsREDQueue hace otras cosas como varios binds y algunas inicializaciones. Al crearse el objeto wfq_hand como tipo WFQdsHandler, a su vez, se reserva espacio en memoria para sus variables (similar a como se hizo con el objeto tipo dsREDQueue referido) y se ejecuta el constructor de WFQdsHandler para wfq_hand. En dicho constructor, el apuntador protegido q de WFQdsHandler, queda apuntando al objeto dsREDQueue creado.

La rutina pública handle(Event *e) de WFQdsHandler corre, a su vez, a la rutina pública WFQdequeueGPS de dsREDQueue (ver dsred.cc modificado) con la instrucción única:

q->WFQdequeueGPS (e);

Esto se puede hacer gracias a que q apunta al objeto dsREDQueue.

Así, el objeto dsREDQueue contiene un objeto wfq_hand (tipo WFQdsHandler) que tiene un apuntador q que apunta al mismo objeto dsREDQueue.

Clase: dsREDQueue
Clase Antecesora: Queue
Grupo de Archivos: dsREDQueue modificado.
Se Declara y Define en Archivo: dsred.h modificado / dsred.cc modificado.

DsREDQueue :: WFQenqueue (Packet *p, int queuid) { ... (ver dsred.cc modificado)

RESUMEN.

ESTA RUTINA CONSIDERA QUE HA LLEGADO UN PAQUETE, APUNTADO POR EL APUNTADOR *p*, A LA COLA *queueid*. SE CALCULA EL TIEMPO VIRTUAL ACTUAL QUE CORRESPONDE AL TIEMPO VIRTUAL DEL EVENTO DE LLEGADA DEL PAQUETE, Y SE PONE EN LA VARIABLE *virt_time*. SE CALCULA EL TIEMPO REAL CORRESPONDIENTE Y SE PONE EN LA VARIABLE *last_vt_update*. SE CALCULA EL TIEMPO VIRTUAL DE FINALIZACIÓN DE ATENCIÓN DEL PAQUETE Y SE PONE EN LA VARIABLE *finish_t[queueid]* SE INTERCALA, EN LAS LISTAS GPS y PGPS, EL ELEMENTO DE CONTROL CORRESPONDIENTE AL PAQUETE QUE ACABA DE LLEGAR, QUE CONTIENE EL TIEMPO VIRTUAL DE FINALIZACIÓN DE ATENCIÓN DEL PAQUETE, *finish_t[queueid*, Y EL NÚMERO DE COLA EN DONDE ESTÁ EL PAQUETE, *queueid*.

SE ACTUALIZA ADECUADAMENTE $\sum_{i \in B_j} \phi_i$ (VARIABLE *sum*) Y EL NÚMERO DE PAQUETES EN ESPERA EN LA COLA DE INTERÉS *queueid*.

EL EVENTO CORRESPONDIENTE A LA PARTIDA FUTURA MÁS CERCANA DE UN PAQUETE DEL SISTEMA GPS, ES EL OBJETO TIPO EVENT APUNTADO POR wfq_event (OBJETO QUE YA ESTABA DADO EN EL INICIO DE ESTA RUTINA) SI DICHO EVENTO NO ES NULO, ENTONCES SE CANCELA DEL SCHEDULER PORQUE SE VA A ACTUALIZAR.

AHORA SE CORRE LA SUBRUTINA dsREDQueue :: WFQscheduleGPS() EN DONDE SE CREA UN NUEVO OBJETO (UN EVENTO) TIPO EVENT, APUNTADO POR wfq_event, Y SE CALCULA EL VALOR DE LA VARIABLE *Next(t)* (QUE INDICA EL TIEMPO REAL CORRESPONDIENTE AL TIEMPO VIRTUAL FUTURO MÁS PRÓXIMO DE FINALIZACIÓN DE ATENCIÓN A UN PAQUETE QUE ESTÁ

GUARDADO EN LA LISTA GPS) CON *Next(t)* SE ASIGNA EN EL SCHEDULER ESTE EVENTO (MANEJADO POR LA VARIABLE *tmp*)

REQUERIMIENTOS.

SE DEBE TENER EL TIEMPO VIRTUAL DEL ÚLTIMO EVENTO, QUE LLEGA EN LA VARIABLE *virt_tim*e (QUE SE REPRESENTA AQUÍ COMO $V(t)$) Y CUYO TIEMPO REAL CORRESPONDIENTE LLEGA EN LA VARIABLE *last_vt_update* (QUE SE REPRESENTA AQUÍ COMO t) SE DEBE TENER EL TIEMPO VIRTUAL DEL EVENTO DE FINALIZACIÓN DE ATENCIÓN DEL ANTERIOR PAQUETE EN LA COLA, QUE LLEGA EN LA VARIABLE *finish_t[queueid]* (QUE SE REPRESENTA AQUÍ COMO F_i^{k-1} DONDE EL NÚMERO DE PAQUETE ANTERIOR ES EL #$k-1$ Y EL NÚMERO DE LA COLA ES #*i*) TAMBIÉN SE REQUIERE LA VARIEBLA TIPO EVENTO wfq_EVENT.

EN LA RUTINA SE EJECUTAN LAS SIGUIENTES ACCIONES.

El tiempo actual real se pone en la variable *now* (se representa aquí en las ecuaciones como $t+\tau$), y el tiempo real del último evento llega en la variable *last_vt_update.*

Se actualiza el tiempo virtual del evento de llegada del paquete a partir del tiempo virtual del evento anterior. Dicho tiempo se encuentra en la variable *virt_time*. En términos de ecuaciones se calcula $V(t+\tau)=V(t)+\frac{\tau}{\sum_{j\in B}\phi_j}$, donde *virt_time* = $V(t)$, variable que se va a actualizar para acabar siendo igual a $V(t+\tau)$ La variable del programa *sum* = $\sum_{j\in B}\phi_j$. Es correcto que *sum* aun no se actualice pues depende de las llegadas de paquetes hasta justo antes de la llegada de este paquete que acaba de llegar. El tiempo que ha pasado entre este evento de llegada y el último evento es τ = *now – last_vt_update*. Todas estas variables son variables protegidas de dsREDQueue.

Se calcula el tiempo virtual finalización de atención del paquete, $F_i^k = S_i^k + {L_i^k}/{R\phi_i}$ (ver fórmula 10 de Parekh y Gallager), donde $S_i^k = \max\left\{F_i^{k-1}, V\left(a_i^k\right)\right\}$ (en esta fórmula se considera que el paquete que ha llegado es el paquete número *i* que ha llegado a la cola *k* desde que la cola dejó de estar inactiva) La variable protegida de dsREDQueue, *finish_t[queueid]*, equivale a F_i^{k-1}, y, con esta fórmula, se va a actualizar para acabar siendo igual a F_i^k .

Se tiene una nota en donde se preguntan si la variable protegida *bandwidth* es realmente una variable que representa bits.

Ahora sí, se actualiza la sumatoria *sum* = $\sum_{j \in B} \phi_j$ (aquí la variable protegida *queueWeight[queueid]* equivale a $\phi_{queueid}$), lo que significa que se le sumará el valor $\phi_{queueid}$, siempre y cuando la variable protegida *B[queueid]* hubiese sido igual a 0, es decir, que la cola *queueid* hubiese estado inactiva. *B[queueid]* es el número de paquetes en espera en la cola *queueid*. Posteriormente se suma 1 a *B[queueid]*.

Se inserta el elemento de control correspondiente al paquete apuntado por *p*, que contiene la información de las variables *queueid* y *finish_t[queueid]*, tanto en la lista PGPS como en la lista GPS, con las rutinas PGPS_list.insert_order, y GPS_list.insert_order, definidas en wfq-list.h.

Dichas listas están formadas de elementos, tipo elem (de estructura elem) de la forma

Ilustración 6

Donde:

El elemento key recibe el valor finish_t[queueid] (variable double que representa el tiempo virtual y representada con la variable k en la rutina insert_order)

El elemento data recibe a la variable queueid (variable int que representa al número de cola física y representada con la variable el en la rutina insert_order)

El elemento next es un apuntador que conetiene la dirección del elemento siguiente hacia atrás (hacia head) (head es el elemento de menor tiempo virtual y se considera el último de la lista – es elemento que se va a tomar en el próximo dequeue –ver WFQdequeuePGPS)

El elemento prev es un apuntador que contiene la dirección de elemento siguiente hacia delante (hacia tail) (tail es el elemento de mayor tiempo virtual y se considera el primero de la lista)

Cada lista (class List) contiene las variables head y tail que son apuntadores a los elementos de la lista head y tail. El elemento tail es aquél que tiene el tiempo virtual (el) más grande de la lista y se considera que es el primer elemento.

Si el apuntador protegido *wfq_event* a un objeto tipo evento, es nulo o cero entonces el evento relacionado se cancela y se borra de memoria.

Se ejecuta la rutina WFQscheduleGPS.

El evento correspondiente a la finalización futura más cercana de un paquete está apuntado por wfq_event (ya estaba desde el inicio de la rutina) Si dicho evento no era nulo, entonces se cancela el evento del Scheduler porque se va a actualizar.

Ahora se crea un nuevo objeto (un evento) tipo Event apuntaod por wfq_event. Se calcula el valor de la variable *Next(t)* (que está en la variable *tmp* y que indica el tiempo real correspondiente al tiempo virtual futuro más próximo de finalización de atención a un paquete que está guardado en la lista GPS) Se asigna en el scheduler dicho evento.

Nota. No hay una cuenta de longitud de colas.

No hay una relación entre los elementos de la lista y los paquetes. No es necesaria esta relación pues los paquetes, en cada cola, están en orden de llegada y se atenderán FIFO. Cuando se va a hacer un dequeuePGPS lo que se hace es seleccionar la próxima cola que va a ser atendida (dequeued) (con la rutina WFQdequeuePGPS) (esta rutina toma el elemento head de la lista y la quita de ella usando las rutinas PTPS_list.get_data_min y PGPS_list. extract) Con la metodología seguida, de forma automática, se mantiene la relación entre cada paquete en las colas y los elementos correspondientes a sus tiempos virtual y real de salida, en las listas.

Clase: dsREDQueue
Clase Antecesora: Queue
Grupo de Archivos: dsREDQueue modificado.
Se Declara y Define en Archivo: dsred.h modificado / dsred.cc modificado.

dsREDQueue :: WFQdequeueGPS (Event *e) { …

RESUMEN.

LA RUTINA RECIBE EL APUNTADOR DE UN EVENTO EN EL SCHEDULER, EL APUNTADOR *e*, QUE ES EL EVENTO CORRESPONDIENTE A LA FINALIZACIÓN FUTURA MÁS CERCANA DE UN PAQUETE, DADO EN TIEMPO REAL.

PARA QUE ESTA RUTINA SUCEDA SE HA QUITADO UN PAQUETE DE COLA. DICHA PARTIDA DEBE CORRESPONDER AL EVENTO APUNTADO POR *e*. LA SIMULACÓN DEBE HABER LLEGADO A DICHO EVENTO EN EL SCHEDULER Y DEBEN HABERSE DISPARADO LAS ACCIONES CORRESPONDIENTES, Y DICHO EVENTO SE DEBE HABER REMOVIDO DEL SCHEDULER (PERO PARECE QUE NO SE BORRÓ DE MEMORIA PORQUE ESTA RUTINA, WFQdequeueGPS, LO BORRA DE MEMORIA)

WFQdequeueGPS EXTRAE EL ELEMENTO DE CONTROL CORRESPONDIENTE AL PAQUETE PROXIMO A SALIR, DE LA LISTA GPS, Y BORRA DICHO ELEMENTO DE DICHA LISTA.

SE ACTUALIZAN ADECUADAMENTE $\sum_{i \in B_j} \phi_i$ (VARIABLE *sum*) Y EL NÚMERO DE PAQUETES B[queueid] EN ESPERA EN LA COLA DE INTERÉS *queueid*.

FINALMENTE SI EL SISTEMA GPS NO ESTA OSCIOSO, SE CORRE LA SUBRUTINA dsREDQueue :: WFQscheduleGPS() EN DONDE SE CREA UN NUEVO OBJETO (UN EVENTO) TIPO EVENT, APUNTADO POR wfq_event, EN DONDE SE VA A PONER EN EL SCHEDULER LA PARTIDA MÁS PRÓXIMA FUTURA, EN TIEMPO REAL, DEL SISTEMA GPS. PARA ESTO, SE CALCULA EL VALOR DE LA VARIABLE *Next(t)* (QUE INDICA EL TIEMPO REAL CORRESPONDIENTE AL TIEMPO VIRTUAL FUTURO MÁS PRÓXIMO DE FINALIZACIÓN DE ATENCIÓN A UN PAQUETE QUE ESTÁ GUARDADO EN LA LISTA GPS) CON *Next(t)* SE ASIGNA EN EL SCHEDULER ESTE EVENTO (MANEJADO POR LA VARIABLE *tmp*)

dsred.h modificado / dsred.cc modificado.

Todos los cambios que se hacen en la modificación de WFQ a dsred.h y dsred.cc, en donde se debe incluir el archivo wfq-list.h (para llevar la lista de elementos de control) no modifican en lo absoluto la actualización de las longitudes de las colas cuando hay encolamientos o desencolamientos.

Operación del Manejador Handler.

wfq_hand.handle(wfq_event)

wfq_hand es el nombre del manejador "handler" del objeto (cola) tipo dsREDQueue.

wfq_event es un apuntador a un objeto tipo evento.

La rutina handle solo tiene la línea

q->WFQdequeueGPS(e) donde e es un apuntador a una variable tipo Event.

q es un apuntador igualado al apuntador this (que apunta a la tipo dsREDQueue en operación)

Cuando se hace wfq_hand.handle(*e) se corre realmente (ver scheduler.cc) this->WFQdequeueGPS(e); (ver llamada a Scheduler en WFQscheduleGPS en dsred.cc)

Variable sobre el Número de Colas en una Cola DiffServ (clase dsREDQueue)

La variable numQueues_ es el número de colas con las que se trabaja. Se debe dar un valor a esta variable entre 1 y el número máximo NUM_QUEUES que es igual a 8 y que está definida así en dsred.h y dsred.h modificado.

Apéndice 4. Modificaciones al Simulador ns para la Operación de WFQ con Pesos que tienen Cambio Dinámico.

Alfredo Mateos modifica el simulador ns en los siguientes términos.

La versión de ns que se modifica es la 2.33 (originalmente la 2.27). Estos cambios sirven para modificar la operación de ns, a fin de que los despachadores WFQ y WRR operen de manera dinámica con respecto a sus pesos. Estos cambios dependerán de las longitudes de las colas.

Los cambios se hacen exclusivamente en los archivos: *dsred.h*, *dsred.cc* y *wfq-list.h* (en C++) A estos archivos se les añadieron rutinas nuevas y modificó rutinas existentes. Todas las adiciones y modificaciones están claramente documentadas en los mencionados archivos.

La forma de hacer el cambio es tomar los archivos *dsred.h*, *dsred.cc* y *wfq-list.h* tal como fueron generados por Mrkaic [2], tal como estaban en su publicación, y cambiarlos por los archivos *dsred.h* y *dsred.cc* que estaban en la versión de ns que tenía Alfredo Mateos (el archivo *wfq-list.h* no existía en la versión que tenía Alfredo Mateos) Hay que hacer notar que las modificaciones de Mrkaic no forman parte oficial de las versiones de ns y solo se pueden encontrar en las ligas ofrecidas en [2].

Ya que la versión de ns que tenía Mrkaic era anterior a aquella con la que trabajó Alfredo Mateos, entonces se hizo una revisión cuidadosa para identificar diferencias en los archivos debido a cambios en las versiones, que tuviesen que corregirse para dejar los archivos compatibles con las nuevas versiones 2.27. Efectivamente se encontraron diferencias por versiones, las que se indican más adelante.

Dado el método anterior, Alfredo Mateos ya no tuvo que copiar las adiciones de Mrkaic a *ns*.

Las rutinas completas indicadas, modificadas por Alfredo Mateos, se incluyen en este trabajo. Asimismo, se incluyen las rutinas correspondientes originales de la versión ns 2.33 (originalmente 2.27), para referencia.

Rutinas Modificadas de *dsred.cc* (*dsred.h*) para Implantar la Operación Dinámica en los Despachadores WFQ y WRR

dsREDQueue::dsREDQueue( )
*void dsREDQueue::WFQenqueue(Packet *p, int queueid) {*
*void dsREDQueue:: WFQdequeueGPS(Event *e) {*
void dsREDQueue::reset( ) {
void dsREDQueue::selectQueueToDeque( ) {
void dsREDQueue::addQueueWeights(int queueNum, int weight) {

*int command(int argc, const char*const* argv)*

Las variables y rutinas añadidas se enlistan a continuación. Las explicaciones de las mismas se pusieron en inglés, por Alfredo Mateos, en el cuerpo de los archivos mencionados.

Variables Añadidas a *dsred.cc* (*dsred.h*) para Implantar la Operación Dinámica en los Despachadores WFQ y WRR

```
double Fmaxtmp; // Maximum virtual time of a weight change process block.
double Fmax; // Maximum virtual time.
double queueWeightpr[MAX_QUEUES]; // Present queue weight. One per queue.
double queueWeightnw[MAX_QUEUES]; // New queue weight. One per queue.
int wchpr; // Weight Change Process. If wchpr = 1 the process is on.
double start_t[MAX_QUEUES]; // Part of formula 11 of Gallager and Pareck
 // article. Used in the WFQenqueue routine of the dsREDQueue class.
int set_chw_op; // If this variable = 1 the option for Weight-change Process
 // is operational. This option has a 0 default value and is changed via OTcl
 // instructions which call the setChgWghtOpt routine. Each dsREDQueue object
 // has its own the set_chw_op variable.
```

Es muy importante observar la variable *set_chw_op* con la cual se puede tener, desde OTcl, comunicación con la parte compilada del simulador, sirve para indicar si se va a utilizar la forma de operación en donde los pesos cambian de forma dinámica, o no.

Rutinas Añadidas a *dsred.cc* (y *dsred.h*) para Implantar la Operación Dinámica en los Despachadores WFQ y WRR

```
double dsREDQueue::getWeightedLength_q(redQueue *q); // Get weighted length of
 // one physical queue.
int dsREDQueue::getRealLength_q(redQueue *q); // Get real length of one physical
 // queue.
double dsREDQueue::getWeightedLength_q_prec(redQueue *q, int); // Get weighted
 // length of one virtual queue of one physical queue.
int dsREDQueue::getRealLength_q_prec(redQueue *q, int); // Get real length
 // of one virtual queue of one physical queue.
double dsREDQueue::get_sum_WeightedLengths(); // Get the sum of weighted
 // lenghts of entire physical queues.
void dsREDQueue::set_new_Weights(); // The weights queueWeightnw[ ] get
 // new weight values.
int dsREDQueue::change_weights(); // If result = 1 the weights must be changed.
void dsREDQueue::setChgWghtOpt(int); // Routine run from the interpreter, for
//   setting the option for the Weight-Change process to operate. It modifies
//   the value of the variable set_chw_op.
// The instruction from Otcl is: "$dsredq setChgWghtOpt 1", to set the
//  Weight-Change Option on (assuming that dsredq is the name of the OTcl variable
```

// that refers to the dsREDQueue object)

Rutinas Añadidas a *wfq-list.h* (y *dsred.h*) para Implantar la Operación Dinámica en los Despachadores WFQ y WRR

double get_key_max( )

Comentarios Importantes sobre algunas Modificaciones Realizadas para Corregir Diferencias en la Versión de Mrkaik con relación a la Versión 2.27 con la que Trabajó Alfredo Mateos.

En *dsred.cc* se dejó puesta la rutina *void applyTSWMeter(Packet *pkt)* con la versión utilizada por Mrkaic [2] (antigua con relación a la versión de ns utilizada por Alfredo Mateos), porque no había influencia en la operación para los usos de este trabajo. El cambio de esta rutina a la forma de la versión usada por Alfredo Mateos hubiese sido bastante fácil y podría hacerse en cualquier momento. La versión utilizada por Alfredo Mateos hubiese contenido un una llamada dicha rutina con diferentes parámetros: *void dsREDQueue::applyTSWMeter(int q_id, int pkt_size)* Esta rutina se llama desde *Packet* dsREDQueue::deque( )*

La versión de Mrkaic tenía, en la rutina *int command(int argc, const char*const* argv)*, en su primera línea if: *if (strcmp(argv[1], "configQ") == 0)*, se hace una llamada a la rutina *config* definida en *dsredq.cc* (Pág. 1), que era anticuada por su parámetros, con relación a la versión de ns utilizada por Alfredo Mateos, así que esta forma de llamada fue actualizada por Alfredo Mateos, en el archivo *dsred.cc*, en donde también se escriben los comentarios al respecto, con la explicación de esta situación.

La siguiente modificación es en extremo importante, y viene de la versión 2.27, de ns (con la que trabajó Alfredo Mateos)

En las versiones 2.33 y 2.27 de ns-2, con las que se trabajó, en el archivo *dsred.h*, en la definición de la clase dsREDQueue (cola tipo dsRED que incluye varias colas RED) se tenía la línea siguiente (donde MAX_QUEUES era una variable de sistema definida para ser igual a 8):

redQueue redq_[MAX_QUEUES]

Cuando se ejecutaban las línea de *dsred.h*, con esta instrucción no solo se estaba declarando que habría una arreglo redq_[8] (redq_[0] a redq_[8]) de objetos de clase redQueue (objeto que define las colas RED), como parte de la clase dsREDQueue, sino que además, por la forma de la línea, se estaba reservando espacio de memoria para estos objetos. Este espacio quedaba reservado para la clase dsREDQueue pero aun no había objetos dsREDQueue que se hubiesen creado. Esto crearía un problema.

Lo anterior no creaba error de compilación, pero sí error en tiempo de ejecución por un "error de acceso a memoria" (Segmentation Fault). Era curioso que el programa ns antes de ser modificado por Alfredo Mateos, no daba problemas, aun existiendo esta situación, pero después de la modificación, estos problemas de tiempo de ejecución de presentaron.

La corrección a este problema se hizo cambiando la instrucción indicada, a la forma:

*redQueue *redqp_[MAX_QUEUES];*

Es decir, ahora en la definición de la clase dsREDQueue se incluía solamente un arreglo de apuntadores de tipo redQueue que aquí se denotan *redqp_* (refiriéndonos con la última *p* a "Pointer"), y desaparecimos las variables *redq_*.

También, en el constructor de la clase dsREDQueue, definido en el archivo *dsred.cc*, casi al final de dicha definición, antes de la última instrucción "*reset()*", se incluyen las líneas en donde se crea espacio de memoria para las ocho variables apuntadoras *redqp_* que forman ya parte de la clase dsREDQueue. Esto se hizo con las siguientes líneas añadidas:

```
 // CHANGE BY ALFREDO MATEOS -- SEE CHANGE IN the declaration of
 //  dsredQueue in dsred.h.
 // Now a number of MAX_QUEUES objects are created, with pointers redqp_[ ]
   for (i = 0; i < MAX_QUEUES; i++) {
    redqp_[i] = new redQueue;
  }
// END OF CHANGE BY ALFREDO MATEOS.
```

Por último, cuando dentro de las operaciones correspondientes a alguna cola dsREDQueue se hiciese una llamada a una rutina o variable de una de sus colas RED (apuntadas por redqp_[0], …, redqp_[7]), dichas llamadas ya no se harían refiriéndose al nombre de dicha cola RED, sino que se haría ahora con llamadas refiriéndose al apuntador de dicha cola RED. Por ejemplo, la línea que antes era:

```
for (i = 0; i < MAX_QUEUES; i++) redq_[i].qlim = limit();
```

Ahora ha cambiado a

```
for (i = 0; i < MAX_QUEUES; i++) redqp_[i]->qlim = limit();
```

Otros Cambios Hechos a los Archivos dsred.cc y dsred.h de ns, en su Versión 2.33 (antes 2.27), para la Implantación del Despachador WFQ.

La Versión de ns utilizada por Mrkaic [2], para incorporar la funcionalidad del despachador WFQ en ns, con relación a las versiones utilizada por A. Mateos de ns, es más antigua. Por esta razón, se hicieron los siguientes cambios.

Se dejó la rutina applyTSWMeter de la versión utilizada por Mrkaic [2]. Dicha rutina es diferente en los archivos dsred.h y dsred.cc.

En la versión ns utilizada por Mrkaic, en el archivo dsred.cc, se tenían las siguientes líneas, en la función dsREDQueue::command.

```
  if (strcmp(argv[1], "configQ") == 0) {
    redq_[atoi(argv[2])].config(atoi(argv[3]), argv);
    return(TCL_OK);
  }
```

Estas líneas fueron sustituidas por las siguientes, para que estuviesen acordes con la versión ns 2.33 y 2.27.

```
  if (strcmp(argv[1], "configQ") == 0) {
    // modification to set the parameter q_w by Thilo
    redq_[atoi(argv[2])].config(atoi(argv[3]), argc, argv);
    return(TCL_OK);
  }
```

En las anteriores líneas, de debe notar que la llamada a la rutina config del objeto redq_[atoi(argv[2])] está cambiada en la versión de Mrkaic.

Casi al final del archivo desred.cc de la versión utilizada por Mrkaic, faltaban rutinas como "getAverageV, y "getCurrentV", las cuales se pusieron.

MODIFICACION al archivo wfq-list.h el 24 de febrero de 2011.

En la clase List , en su rutina insert_order(X el, double k) se hace la siguiente modificación.

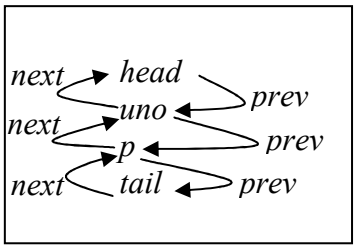
En lugar de:

```
if(p->next !=) {
 p->next->prev = tmp;
 tmp->next = p->next;
} else
```

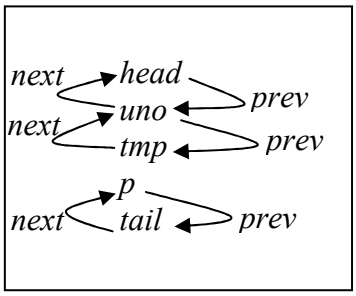
Se pone

```
if(p->next !=) {
 (p->next)->prev = tmp;
  tmp->next = p->next;
  p->next = tmp;
  tmp-prev = p;
} else
```

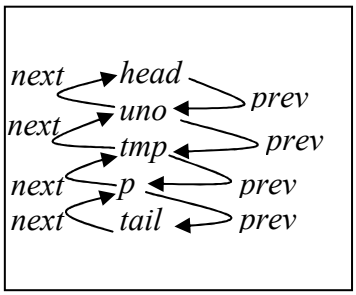
Notar que se arreglan unos paréntesis. Asimismo, lo anterior es importante porque al acomodar un nuevo elemento en una lista (elemento apuntado con el apuntador tmp, el cual en las anteriores líneas se va a poner arriba del apuntador p, pero se carecía de ligas entre los elementos, lo que debía completarse. Para ejemplificar, digamos que el elemento que está arriba de aquél apuntado por el apuntador *p* está apuntado por el apuntador *uno*. El primer apuntador es *head* y el último es *tail* y digamos que ni uno ni p coinciden con *head* y *tail* respectivamente. Así que el siguiente cuadro ejemplifica esquemáticamente las posiciones:



Va a entrar el nuevo elemento apuntado por *tmp* que se supone que quedará arriba de p. Según lo que se tenía en el programa quedaba lo siguiente.

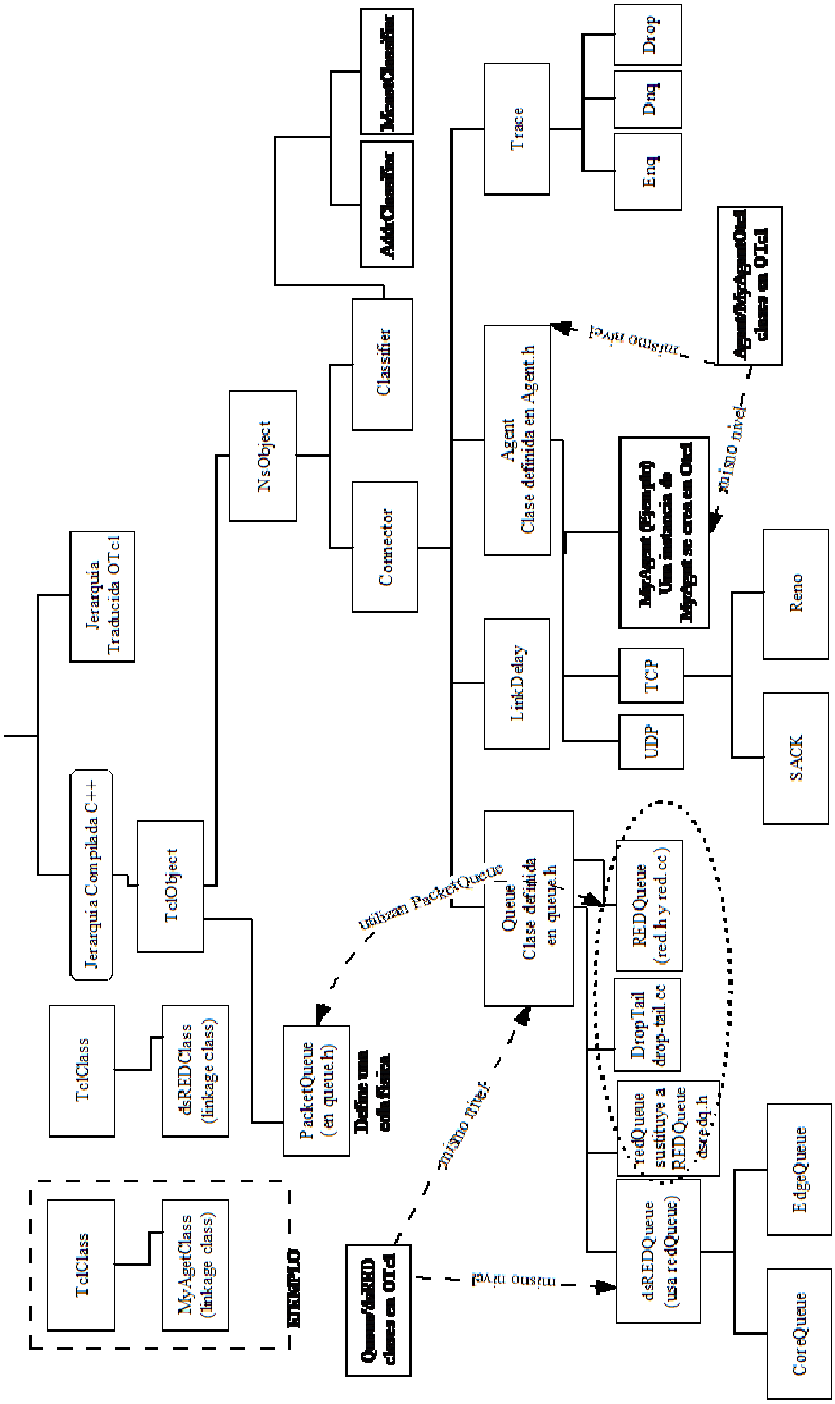


Entonces las líneas adicionales de corrección agregan los faltantes para que quede lo siguiente:



Estructura de las Clases en la Programación del Simulador ns.

En el siguiente diagrama se puede observar una jerarquía y clasificación de las "clases", dada la programación orientada a objetos utilizada para construir el Simulador ns.



Apéndice 5. Configuraciones Adicionales.

Notas Sobre la Configuración de ns en Linux.

INICIALMENTE, al hacer modificaciones en los archivos en el directorio /usr/local/ns-allinone-2.27/ns-2.27/diffserv, desde el directorio /usr/local/ns-allinone-2.2.7/ns-2.27, se estaba corriendo la rutina ./configure.

Lo anterior no funcionaba bien así que nuevamente se instaló todo el ns corriendo la instrucción ./install nuevamente, estando en el directorio /usr/local/ns-allinone-2.27, y todo se instaló correctamente nuevamente. Al final de la instalación se tuvieron las siguientes notas (las anotaciones para la versión 2.33 son muy similares).

```
Please compile your gt-itm & sgb2ns separately.
Ns-allinone package has been installed successfully.
Here are the installation places:
tcl8.4.5:       /usr/local/ns-allinone-2.27/{bin,include,lib}
tk8.4.5:                /usr/local/ns-allinone-2.27/{bin,include,lib}
otcl:           /usr/local/ns-allinone-2.27/otcl-1.8
tclcl:          /usr/local/ns-allinone-2.27/tclcl-1.15
ns:             /usr/local/ns-allinone-2.27/ns-2.27/ns
nam:    /usr/local/ns-allinone-2.27/nam-1.10/nam
xgraph: /usr/local/ns-allinone-2.27/xgraph-12.1

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

Please put /usr/local/ns-allinone-2.27/bin:/usr/local/ns-allinone-
2.27/tcl8.4.5/unix:/usr/local/ns-allinone-2.27/tk8.4.5/unix into your
PATH environment; so that you'll be able to run itm/tclsh/wish/xgraph.

IMPORTANT NOTICES:

(1) You MUST put /usr/local/ns-allinone-2.27/otcl-1.8, /usr/local/ns-
allinone-2.27/lib,
    into your LD_LIBRARY_PATH environment variable.
    If it complains about X libraries, add path to your X libraries
    into LD_LIBRARY_PATH.
    If you are using csh, you can set it like:
                setenv LD_LIBRARY_PATH <paths>
    If you are using sh, you can set it like:
                export LD_LIBRARY_PATH=<paths>

(2) You MUST put /usr/local/ns-allinone-2.27/tcl8.4.5/library into your
TCL_LIBRARY environmental
    variable. Otherwise ns/nam will complain during startup.

(3) [OPTIONAL] To save disk space, you can now delete directories
tcl8.4.5
    and tk8.4.5. They are now installed under /usr/local/ns-allinone-
2.27/{bin,include,lib}
```

```
After these steps, you can now run the ns validation suite with
cd ns-2.27; ./validate

For trouble shooting, please first read ns problems page
http://www.isi.edu/nsnam/ns/ns-problems.html. Also search the ns mailing
list ar chive
for related posts.

[root@localhost ns-allinone-2.27]#
```

Con lo anterior, se hicieron las modificaciones al archivo etc/profile (que sirve para la clave "root") de la siguiente forma:

```
pathmunge /sbin
      pathmunge /usr/sbin
      pathmunge /usr/local/sbin
        pathmunge /usr/local/ns-allinone-2.27/bin
        pathmunge /usr/local/ns-allinone-2.27/tcl8.4.5/unix
        pathmunge /usr/local/ns-allinone-2.27/tk8.4.5/unix
        pathmunge /usr/local/ns-allinone-2.27/otcl-1.8
        pathmunge /usr/local/ns-allinone-2.27/lib
```

Para modificar el PATH del usuario individual apiero modifiqué el archivo .bash_profile en el directorio /home/apiero (normalmente invisible si en la terminal se pone ls, se debe usar ls - a) Así que para editarlo se debe escribir en la Terminal: gedit .bash_profile

Se modificó este archivo para que tuviese las líneas:

```
PATH=$PATH:$HOME/bin
PATH=$PATH:"/usr/local/ns-allinone-2.27/bin":"."
PATH=$PATH:"/usr/local/ns-allinone-2.27/tcl8.4.5/unix"
PATH=$PATH:"/usr/local/ns-allinone-2.27/tk8.4.5/unix"
PATH=$PATH:"/usr/local/ns-allinone-2.27/otcl-1.8"
PATH=$PATH:"/usr/local/ns-allinone-2.27/lib"
```

BIBLIOGRAFIA.

[1] A. Parekh, R. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single Node Case", IEEE / ACM Transactions on Networking, 1(3) (1993) 344-357.

[2] A. Mrkaic, (tutor U. Fiedler, supervisor. Prof. Dr. B. Plattner), "Porting a WFQ Scheduler into ns-2's Diffserv Environment", Computer Engineering and Networks Laboratory (Insitut für Technische Informatik und Kommunikationsnetze) Swiss Federal Institute of Technology (Eidgenössische Technische Hochschule Zürich), (2001).

[3] P. Pieda, J. Ethridge, M. Baines, F. Shallwani, "A Network Simulator Differentiated services Implementation", Open IP, Nortel Networks, (2000), Available: http://www-sop.inria.fr/members/Eitan.Altman/COURS-NS/DOC/DSnortel.pdf, Jun 2010.

[4] ns-2 **network simulator. Available: http://nsnam.isi.edu/nsnam/index.php/Main_Page, Jun 2010.**