

# Scheduling

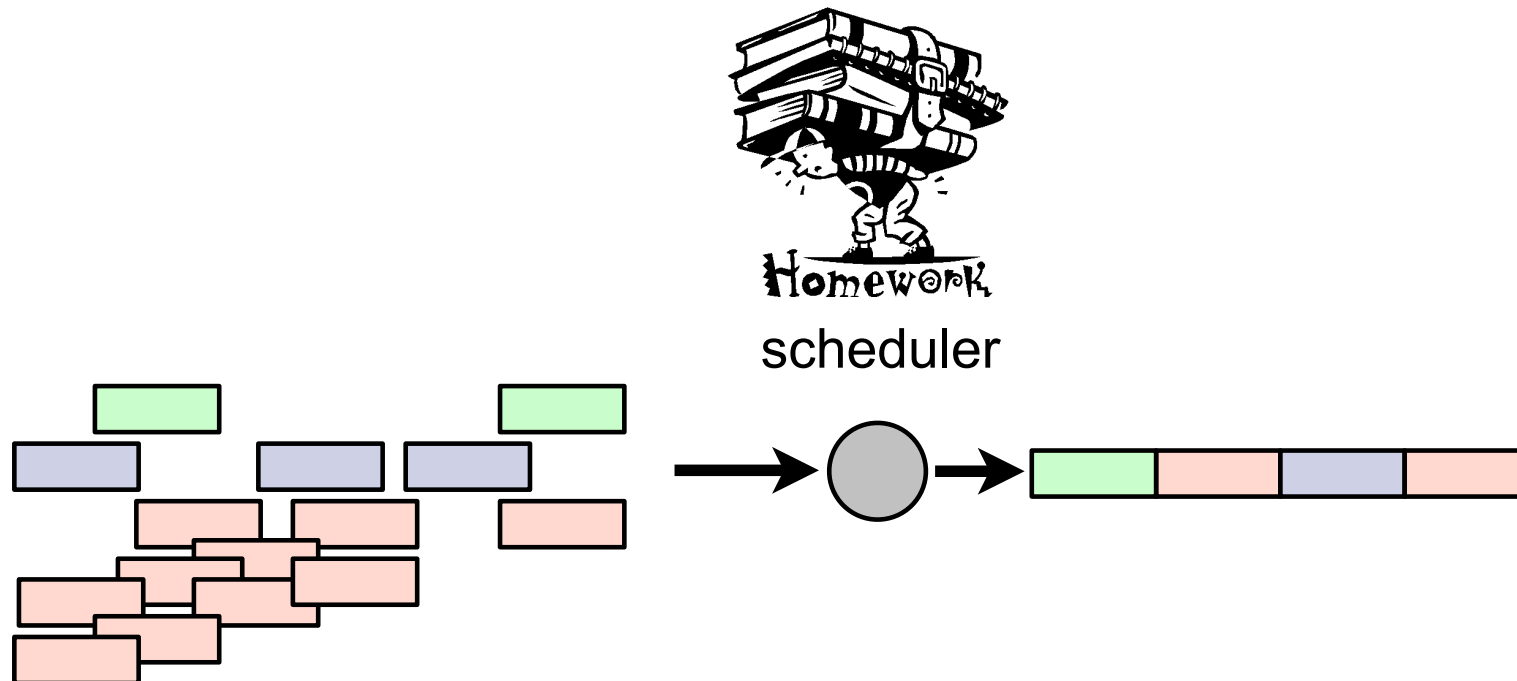
Area de Ingeniería Telemática  
<http://www.tlm.unavarra.es>

Grado en Ingeniería en Tecnologías de  
Telecomunicación, 3º

# *Scheduling: FCFS*

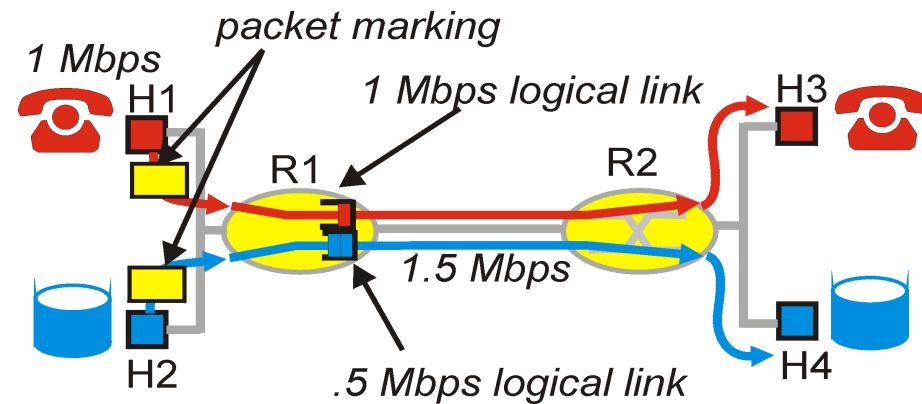
# Scheduling

- Permite compartir recursos
- Emplea una disciplina de planificación para decidir la siguiente petición a atender
- Puede tener lugar en diferentes niveles de una pila de protocolos
- Por ejemplo en el nivel de aplicación sería necesario para decidir la siguiente petición a un servidor que atender
- (...)

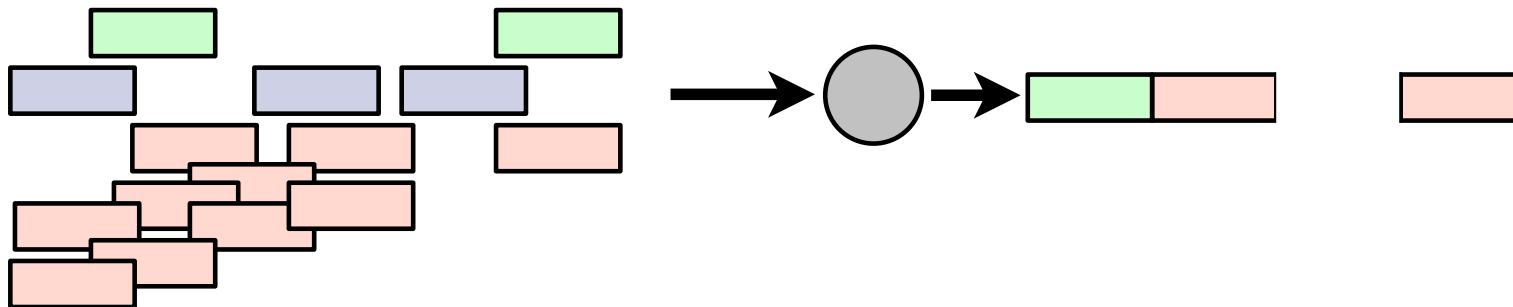


# Scheduling

- Nos centraremos en compartir la capacidad de un enlace
- Y en planificadores **conservativos en trabajo** (*work conserving*): están inactivos solo si la cola está vacía



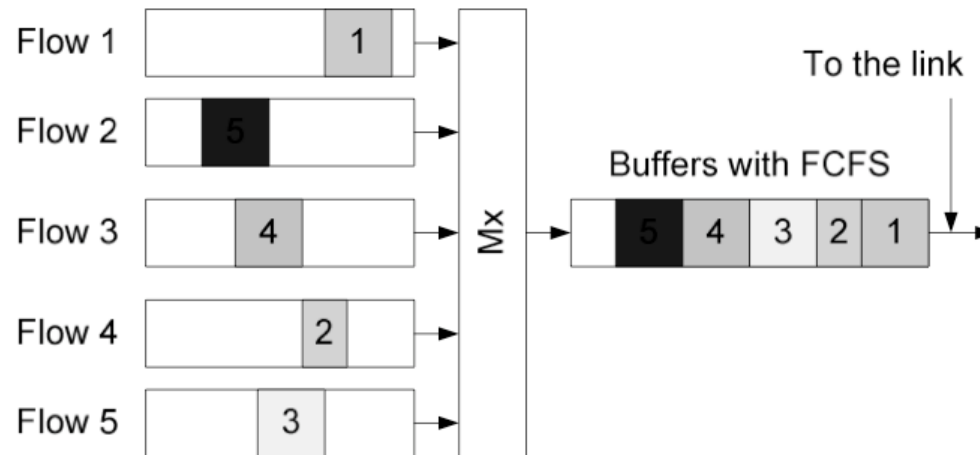
scheduler





# FCFS (FIFO)

- Orden de llegada
- Es el método más rápido y sencillo de implementar
- Se suele utilizar por defecto (*Best Effort*)
- ¿Problemas? (...)



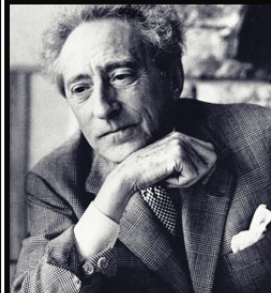
# FCFS (FIFO)

## Problemas

- Limitado por la capacidad del buffer ante congestión (normalmente en número de paquetes)



- No permite diferenciar entre distintos tipos de paquete
- Se logra asignación proporcional a la demanda
- Una fuente *greedy* puede capturar el enlace



Un egoísta es aquel que se empeña en hablarte de sí mismo cuando tú te estás muriendo de ganas de hablarle de ti.

(Jean Cocteau)

# *Scheduling: FCFS*

# *The conservation law*

# The Conservation Law

- La disciplina FCFS no distingue entre diferentes flujos
- FCFS por ejemplo no permite menor retardo a paquetes de un flujo

## Conservation Law

- Nos dice que una disciplina de planificación solo puede mejorar el retardo medio de un flujo frente a FCFS a costa de empeorar el de otro flujo
- (...)



# The Conservation Law

## Ejemplo

- Sea un conjunto de  $N$  flujos en un planificador
- Para el flujo  $i$  la tasa media de llegadas por unidad de tiempo es  $\lambda_i$
- El tiempo medio de servicio de los paquetes del flujo  $i$  es  $x_i$
- La utilización media del enlace debido al flujo  $i$  es  $\rho_i = \lambda_i x_i$
- El tiempo medio de espera en cola de los paquetes del flujo  $i$  es  $q_i$
- Si el planificador es conservativo en trabajo (*work-conserving*) entonces

$$\sum_{i=1}^N \rho_i q_i = \text{Constante}$$

- Es independiente del planificador en concreto



# The Conservation Law

- Un STM-1
- Dos PVCs ATM
  - A. Tasa de llegadas de 10Mbps
  - B. Tasa de llegadas de 25Mbps
- Con FCFS ambos sufren un retardo medio en cola de 0.5 ms
- Con un planificador diferente los paquetes del flujo A sufren un retardo medio en cola de 0.1 ms
- ¿Cuál es el retardo medio en cola que sufren los paquetes del flujo B?

$$\sum_{i=1}^N \rho_i q_i = \text{Constante}$$

- Todas las celdas de igual tamaño así que igual al tamaño medio de paquete:

$$\rho_i = \lambda_i x_i = 10\text{Mbps} / (53 \times 8\text{bits}) \times (53 \times 8\text{bits}) / 155\text{Mbps} = 10/155$$

$$10 / 155 \times 0.5 + 25 / 155 \times 0.5 = 10 / 155 \times 0.1 + 25 / 155 \times R_B$$

$$R_B = 0.66 \text{ ms}$$

# The Conservation Law

- Es decir: para reducir el retardo medio de una clase debemos aumentar el de otra(s)

$$\sum_{i=1}^N \rho_i q_i = \text{Constante}$$



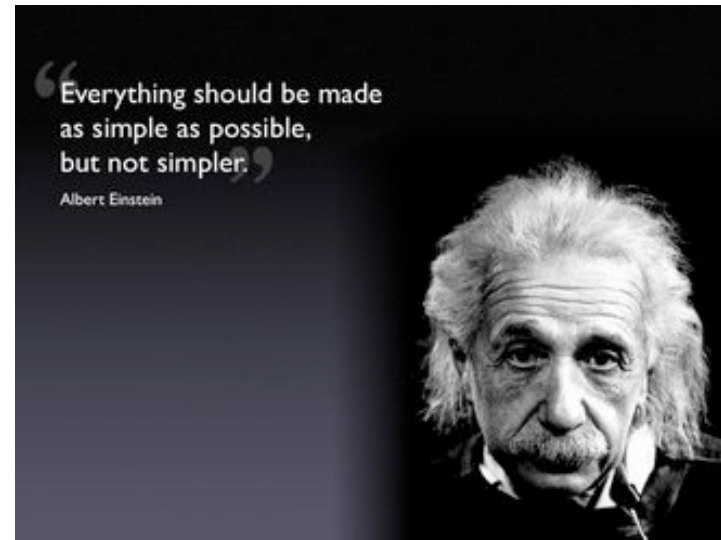
# Características de un planificador

# Características deseables

- **Sencillo de implementar**
- ***Performance bounds* (deterministas o estadísticos)**
- **Que permita implementar un CAC simple**
- **Reparto justo y protección**

# Sencillez

- Que requiera pocas operaciones para ser rápido
- Implementable en hardware
- Que el número de operaciones sea independiente del número de flujos a planificar



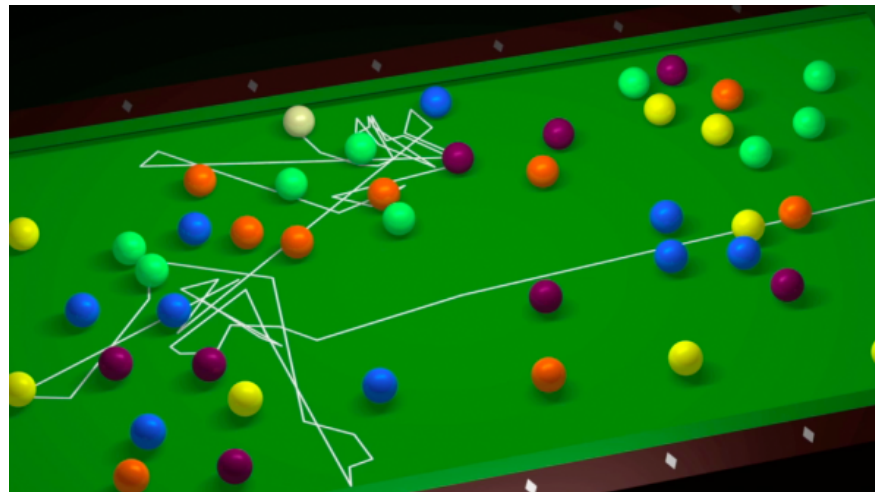
# *Performance bounds*

- Debería permitir garantizar límites (*bounds*) a un flujo
- Bounds extremo a extremo, lo cual implica a todos los schedulers en el camino
- Más simple si emplean la misma disciplina de planificación
- (...)



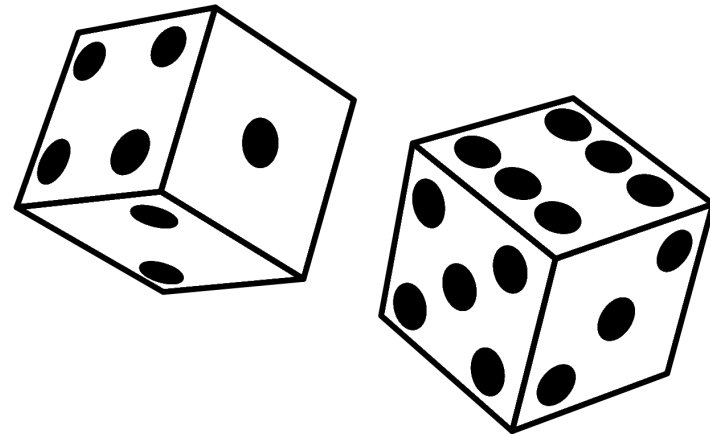
# *Performance bounds*

- **Deterministas**
  - Se cumplen para todos los paquetes del flujo
  - Ejemplo: 10s como cota máxima al retardo extremo a extremo, implica que todos los paquetes sufrirán un retardo menor que 10s
- (...)



# Performance bounds

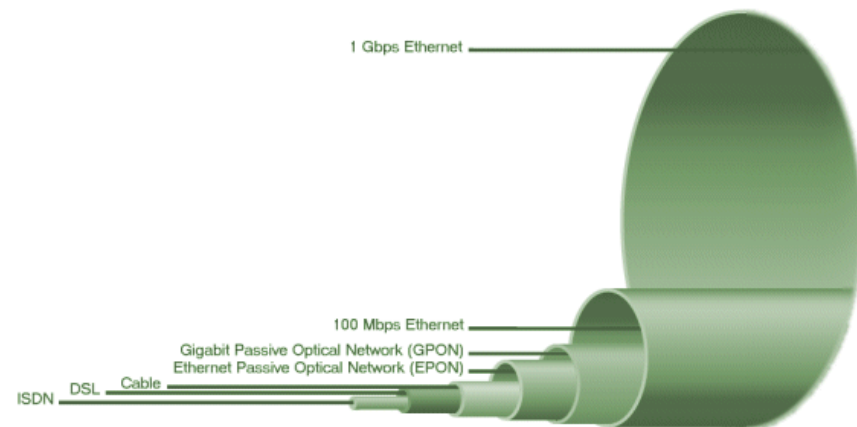
- **o Estadísticos**
  - Probabilístico
  - Ejemplo: menos de 10s de retardo con probabilidad 0.99, implica que la probabilidad de que un paquete sufra más de 10s de retardo es menor del 1%
  - Otra posibilidad es en forma de “uno entre N”
  - Ejemplo: No más de 1 entre 100 paquetes sufrirán un retardo de más de 10s
  - Este caso es más fácil de verificar pero más difícil de implementar pues los conmutadores deben seguir el estado de cada flujo
- Los deterministas suelen requerir más recursos pues son para todos los paquetes



# *Performance parameters*

## **Bandwidth**

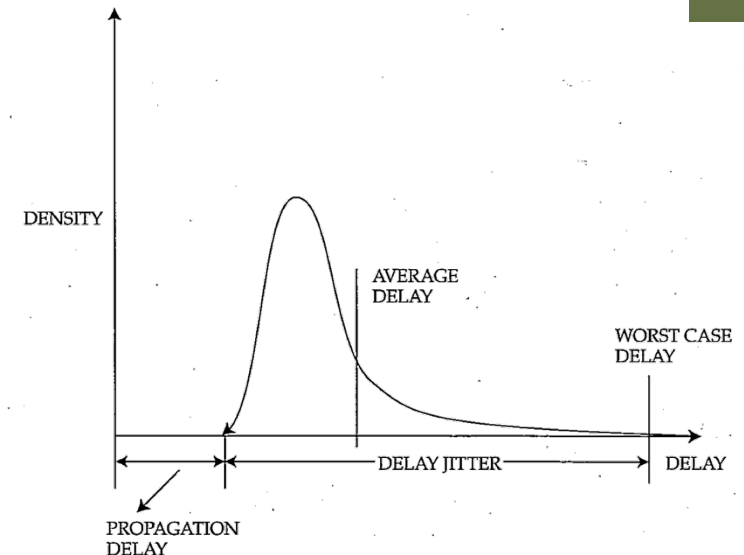
- Recibir al menos un mínimo, medido en un intervalo
- Es el más habitual en las implementaciones
- (...)



# Performance parameters

## Delay

- Determinista o estadístico
- Caso peor: suponer que el resto de flujos se comportan de la peor manera posible
- Medio:
  - Debe ser la media para cualquier patrón de llegadas
  - O sea, imposible de medir
  - Eso lo hace difícil de garantizar
  - Normalmente hablaremos de la media en la duración de un flujo
  - Suele ser una aproximación si dura suficiente y el flujo es independiente del resto
- (...)

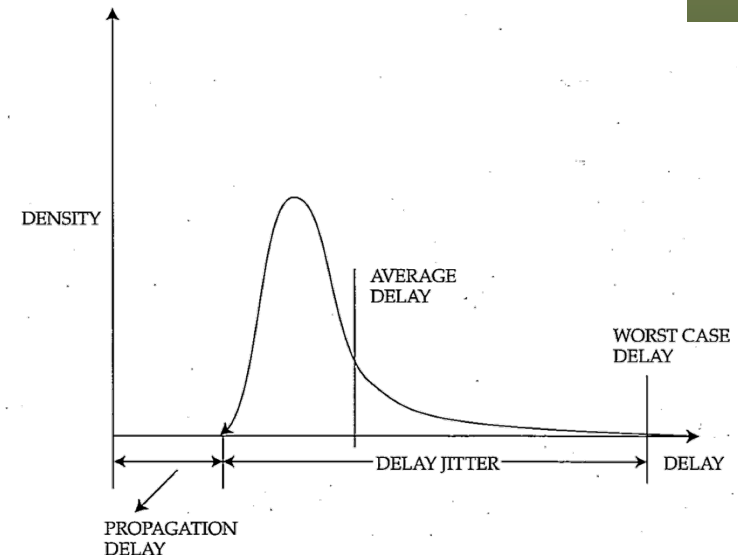




# Performance parameters

## Delay

- Percentil:
  - Por ejemplo que el 99% de los paquetes sufran menos de un retardo
  - De nuevo medible solo para un flujo en concreto
  - Algunos schedulers tienen una dependencia entre BW y delay de forma que para lograr bajo delay hace falta alto BW



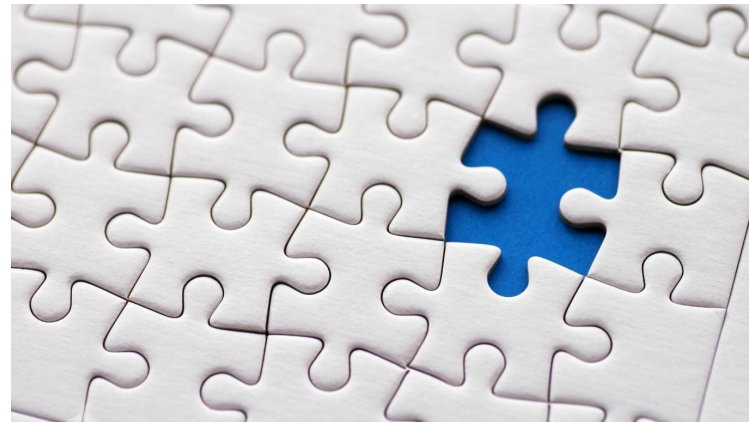
# *Performance parameters*

## **Delay-jitter**

- Acotar la diferencia entre el mayor y el menor retardo posible

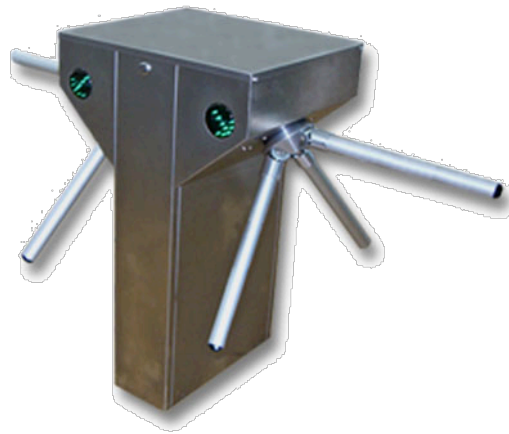
## **Losses**

- Fracción de paquetes (del flujo)



# CAC simple

- Dado el conjunto de flujos existentes y sus requisitos y el nuevo flujo y los suyos
- Decidir si se pueden alcanzar los requisitos del nuevo flujo sin violar los de los anteriores
- Hacerlo sin infrautilizar la red (no aceptando apenas flujos será fácil cumplir los objetivos pero infrautilizamos la red)



# Reparto justo (*max-min fair*)

# Reparto justo y protección

- Flujos con requerimientos estrictos deben tenerlos garantizados independiente de esta “justicia”
- Reparto justo es importante para flujos best-effort
- Reparto decimos que es justo si satisface un *max-min fair share*



- Scheduling es normalmente una decisión local al nodo de conmutación pero la “justicia” para un flujo es un objetivo global
- (...)



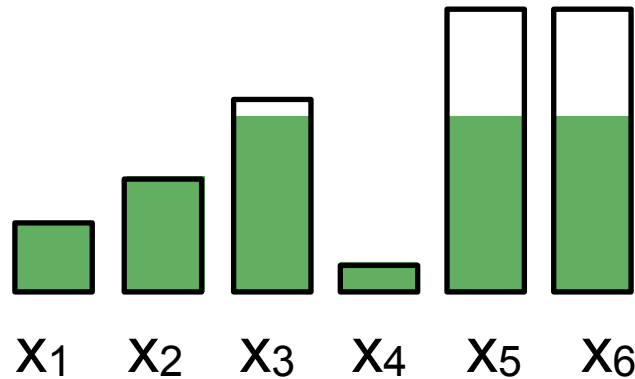
# Reparto justo y protección

- Lograr justicia global con flujos cambiantes no es tan sencillo
- La protección implica que un flujo que envíe más que su asignación justa no afecte al resto
- Un planificador que haga un reparto justo ofrece protección
- (...)



# Reparto justo (max-min fair)

- Para dividir recursos entre un conjunto de usuarios, todos con iguales “derechos” pero diferentes demandas
- De forma simple: a los que piden “poco” se les da lo que piden y lo que sobra se reparte entre los que piden “mucho”
- Formalmente:
  - Asignar recursos en orden creciente de demanda
  - Ningún cliente recibe más de lo que solicita
  - Aquellos cuya demanda no se pueda satisfacer se reparten el remanente del recurso



# Reparto justo (max-min fair)

- Flujos  $1, \dots, n$
- Demandas  $x_1, \dots, x_n$
- Demandas ordenadas  $x_1 \leq x_2 \leq \dots \leq x_n$
- Capacidad a repartir  $C$
- Inicialmente asignar  $C/n$  al flujo 1
- (...)





# Reparto justo (max-min fair)

- Flujos  $1, \dots, n$
- Demandas  $x_1, \dots, x_n$
- Demandas ordenadas  $x_1 \leq x_2 \leq \dots \leq x_n$
- Capacidad a repartir  $C$
- Inicialmente asignar  $C/n$  al flujo 1
- Si esto es más que lo que necesita ( $C/n > x_1$ ) lo que sobra se repartirá entre el resto
- Asignar al flujo 2:  $C/n$  más la parte que le corresponde de lo que sobró del flujo 1, es decir:

$$C/n + \frac{C/n - x_1}{n-1}$$

- Esto puede ser más que lo que el flujo 2 necesita, así que lo que sobra se puede repartir entre el resto
- *Al final todos tienen lo que han pedido o si no había suficiente para eso no tienen menos que cualquier otra fuente que ha pedido más*



# Max-min Fair (Ejemplo)

- Recurso: 10
- Demandas: 2, 2.6, 4 y 5
- $10/4 = 2.5$ 
  - Demasiado para el primer cliente
  - Asignarle 2 y queda 0.5
- Ese 0.5 repartirlo entre los otros 3:
  - $0.5/3 = 0.16666\dots$
  - Asignaciones [2, 2.66, 2.66, 2.66]
  - Demasiado para el segundo cliente
  - Asignarle 2.6 y quedan 0.0666....
- Repartir ese 0.0666... entre los otros 2:
  - $(2.5+0.5/3-2.6)/2 = 0.03333\dots$
  - Asignaciones [2, 2.6, 2.7, 2.7]

# Reparto justo con pesos (*max-min weighted fair*)

# Max-min *Weighted* Fair Share

- En ocasiones se desea una asignación preferente a unos flujos frente a otros
- Se asocia esta preferencia con unos pesos  $w_1, w_2, \dots, w_n$
- Extensión:
  - Los recursos se asignan en orden de demanda creciente, normalizada por el peso
  - Ningún cliente recibe más de lo que solicita
  - Aquellos cuya demanda no se pueda satisfacer se reparten el remanente del recurso en proporción a sus pesos



# Max-min WFS (Ejemplo)

- Recurso: 20. Demandas: 4, 2, 10 y 8. Pesos: 2.5, 4, 0.5 y 1
- Normalización de los pesos:
  - Que el menor valga 1
  - Pesos normalizados: 5, 8, 1 y 2
- En vez de 4 clientes es como si hubiera  $5+8+1+2 = 16$
- $C/n = 20/16 = 1.25$
- La asignación a cada uno sería:
  - $(5 \times 1.25 =) 6.25$ ,  $(8 \times 1.25 =) 10$ ,  $(1 \times 1.25 =) 1.25$  y  $(2 \times 1.25 =) 2.5$
  - El cliente 1 obtiene 6.25 pero solicitaba 4 luego sobra 2.25
  - El cliente 2 obtiene 10 pero solicitaba 2 luego sobra 8
  - El cliente 3 obtiene 1.25 pero solicita 10 (insuficiente)
  - El cliente 4 obtiene 2.5 pero solicita 8 (insuficiente)
- Ha sobrado  $2.25 + 8 = 10.25$  a repartir entre los clientes 3 y 4
  - Sus pesos ya están normalizados (1 y 2).  $C/n = 10.25 / 3 = 3.417$
  - El cliente 3 obtiene 3.417 adicional, en total  $1.25 + 3.417 = 4.667$  (insuficiente)
  - El cliente 4 obtiene 6.834 adicional, en total  $2.5 + 6.834 = 9.334$ , sobra 1.334
  - Lo que sobra del cliente 4 se asigna al cliente 3 y así recibe  $4.667 + 1.334$
- Asignación final: 4, 2, 6, 8