Capítulo 16

Calidad de servicio

Se dice de IP que es un protocolo *best effort*, que es una forma bonita para decir que hace lo que puede, pero podría ser nada en absoluto. La integridad, el orden o la propia entrega son prestaciones que IP consigue eventualmente, pero no están en absoluto aseguradas. Aunque por suerte TCP sí ofrece estas garantías, vamos a ver que hay otros aspectos que no puede resolver. Una de las limitaciones clave de la tecnología de conmutación de paquetes en la que se basa IP e Internet, es que no dispone de control de admisión, es decir, nunca niega el servicio a un nuevo usuario ni rechaza establecer una nuevo flujo a pesar de que la interred esté sufriendo ya una carga excesiva o incluso congestión.

El control de admisión es el aspecto más característico de las redes de conmutación de circuitos (como la telefonía). La red telefónica puede cuantificar la cantidad de recursos disponibles y rechazar el servicio solicitado por un cliente si estima que no lo va a poder satisfacer adecuadamente: el clásico «todas las líneas están ocupadas». El impacto del control de admisión es muy obvio y fácil de entender, pero veremos que hay otras formas de mejorar el servicio.

La Calidad de Servicio (QoS) engloba los mecanismos y tecnologías que intentan garantizar determinado nivel en los parámetros de tráfico para los servicios que ofrece una red de conmutación de paquetes. La idea que subyace a la QoS es que la red debería asignar sus recursos en función de las necesidades de cada usuario, flujo o servicio, bajo la premisa que no todos necesitan las mismas prestaciones. Por ejemplo, una llamada de voz requiere baja latencia, pero puede aceptar cierta pérdida de mensajes, mientras que la descarga de un archivo puede tolerar alta latencia, pero la pérdida de datos es inadmisible.

Los parámetros básicos de la QoS son tasa de transferencia, latencia, *jitter* y tasa de pérdida de paquetes. Empezaremos este capítulo estudiando estos parámetros, su importancia, causas y consecuencias, la forma de medirlos y las técnicas que permiten controlarlos.

16.1. Tasa de transferencia

El primer parámetro que vamos a estudiar es la tasa de transferencia, que mide la velocidad con la que se mueven los datos ya sea a través de una red, de un enlace, un flujo UDP, una conexión TCP, etc. La tasa se mide en bits por segundo (b/s o bps) o bytes por segundo (B/s) y sus múltiplos (ver § D.3).

Se distingue habitualmente entre dos tipos de tasa:

Throughput
: o tasa bruta, es la cantidad total de datos transmitidos en un período de tiempo. Además de la carga útil, incluye cabeceras, reconocimientos, retransmisiones o cualquier otro tipo de información de control; todo lo que llamamos sobrecarga de protocolos.

Goodput
: o tasa neta, considera solo los datos efectivos, la carga útil procedente del usuario o la aplicación.

En realidad ambas tasas se pueden medir en distintas capas, considerando diferentes cabeceras y sus mecanismos de control. La tasa bruta de más bajo nivel debería contabilizar hasta las cabeceras de enlace, por lo que tiene que medirse directamente desde la interfaz de red. Veremos más adelante algunas herramientas, como el analizador de tráfico, que puede medirla. FIXME(tshark, ss).

La tasa neta se suele medir sobre el nivel de transporte, es decir, con los datos que se envían o reciben con las primitivas del socket (como veremos en los ejemplos de código) o sobre el nivel de aplicación, que sería el caso en el que únicamente se cuentan los datos relevantes para el usuario, como por ejemplo, un fichero que se está descargando con HTTP.

También es útil distinguir la tasa en los extremos y en la red:

- **Tasa de envío** (T_s), es la velocidad con la que el proceso emisor entrega datos al SO a través de un socket, que como sabemos, en el caso de TCP no implica necesariamente que esos datos estén saliendo a la red y puede que simplemente se estén almacenando en el buffer de envío.
- **Tasa de recepción**[1] (T_r), es la velocidad con la que el proceso receptor recoge datos a través de un socket, que en realidad implica leer datos del buffer de recepción de la conexión.

[1] en inglés: *drain rate*

- **Tasa de red**[2] (T_n), es la velocidad a la que los datos se mueven a través de la red. Esta tasa no se puede medir directamente, pero se puede estimar a partir de las tasas de envío y recepción.

Lógicamente para cualquiera de las tres pueden ser útiles tanto la tasa bruta como la neta. Los datos se envían en bloques (segmentos o datagramas) que pueden tener tamaños distintos y que son enviados/recibidos a intervalos posiblemente irregulares. Eso condiciona el cálculo y da lugar a distintos métodos. Veamos los más habituales.

16.1.1. Tasa instantánea

Es el cálculo más sencillo, y probablemente también el menos útil. Consiste en contabilizar únicamente los datos transmitidos desde la medición anterior considerando el tiempo transcurrido desde entonces. Formalmente:

$$R_{\text{inst}} = \frac{\Delta B}{\Delta t} \tag{16.1}$$

donde: R_{inst} es la tasa instantánea (bytes por segundo), ΔB es la cantidad de datos transmitida en el último bloque (bytes), y Δt es el tiempo transcurrido desde el envío del bloque anterior (segundos).

La tasa instantánea puede variar drásticamente de un mensaje al siguiente. Por eso se dice que es muy «sensible al ruido», y por ello raramente resulta útil.

Por ejemplo, en Python la tasa de envío neta instantánea podría calcularse como aparece en el siguiente fragmento de código[3]. Fíjate que usamos `time.monotonic()` en lugar de `time.time()` para que el cálculo de los delta no se vea afectado por ajustes en el reloj del sistema.

```
while 1:
    last_time = time.monotonic()
    data = get_data()
    sock.sendall(data)
    byte_rate = len(sent) / time.monotonic() - last_time
    print(byte_rate)
```

16.1.2. Promedio acumulado (CA)

En el otro extremo de la tasa instantánea está el promedio total acumulado o CA (Cumulative Average). Consiste simplemente en calcular la media de toda la transmisión hasta el momento, es decir, el total de datos transmitidos partido por el tiempo transcurrido desde el inicio.

[2] en inglés: *network rate*

[3] Adaptarlo a la tasa de recepción es directo

$$R_{\text{avg}}(t) = \frac{B(t) - B(t_0)}{t - t_0} \tag{16.2}$$

donde: $R_{\text{avg}}(t)$ es la tasa media acumulada (bytes por segundo), $B(t)$ es la cantidad total de datos transmitida hasta ahora (bytes), $B(t_0)$ es la cantidad total de datos transmitida en el instance 0, y $t - t_0$ es el tiempo transcurrido desde el comienzo (segundos).

Lógicamente sufre del efecto contrario a la tasa instantánea. No se adapta en absoluto a las variaciones (nada reactivo), por lo que una reducción o aumento momentáneo de la tasa queda completamente enmascarado.

En Python:

```
start_time = time.monotonic()
sent = 0
while 1:
    data = get_data()
    sock.sendall(data)
    sent += len(data)
    elapsed = time.monotonic() - start_time
    byte_rate = sent / elapsed
    print(byte_rate)
```

LISTADO 16.1: Promedio acumulado

16.1.3. Media móvil simple (SMA)

Una técnica intermedia que ofrece información actualizada, pero más estable que la tasa instantánea, es la 'media móvil simple' o SMA (Simple Moving Average). Se basa en definir una ventana de tiempo que abarca solo los w segundos previos considerando únicamente los datos transmitidos en ese lapso.

$$R_{\text{SMA}}(t) = \frac{B(t) - B(t - w)}{w} \tag{16.3}$$

donde: $R_{\text{SMA}}(t)$ es la tasa media móvil simple (bytes por segundo), $B(t)$ es la cantidad total de datos transmitida hasta ahora (bytes), $B(t - W)$ es la cantidad total de datos transmitida hace w segundos, y w es el tamaño de la ventana (segundos).

Aunque es una medida mucho más reactiva que el promedio acumulado, los nuevos datos tardan algún tiempo ($w/2$ segundos) en reflejarse en la tasa.

En Python:

```
class SMA_RateMeter:
    def __init__(self, window_size=5):
        self.window_size = window_size
        self.window = collections.deque()

    def update(self, sent_bytes):
        now = time.monotonic()
        self.window.append((now, sent_bytes))

        # Eliminar datos fuera de la ventana
        while self.window and (now - self.window[0][0]) > self.window_size:
            self.window.popleft()

    @property
    def rate(self):
        if not self.window:
            return 0

        window_bytes = sum(b for t, b in self.window)
        elapsed = self.window[-1][0] - self.window[0][0]
        return window_bytes / elapsed if elapsed > 0 else 0

sma = SMA_RateMeter(window_size=5)
while 1:
    data = get_data()
    sock.sendall(data)
    sma.update(len(data))
    print(sma.rate)
```

16.1.4. Media móvil exponencial (EMA)

SMA también tiene un problema: los datos nuevos de la ventana tienen el mismo peso que los antiguos, por lo que la tasa no refleja bien las variaciones en el tráfico (poco reactivo). Para paliar ese efecto se puede usar una 'media móvil exponencial', o EMA (Exponential Moving Average), que aplica un decaimiento exponencial, asignando menos peso a los datos más antiguos.

Se calcula utilizando la siguiente fórmula:

$$R_{\text{EMA}}(t) = \alpha \cdot R_{\text{inst}}(t) + (1 - \alpha) \cdot R_{\text{EMA}}(t-1) \tag{16.4}$$

donde: $R_{\text{EMA}}(t)$ es la tasa media móvil exponencial, $R_{\text{inst}}(t)$ es la tasa instantánea en el instante t, y α es el factor de suavizado ($0 < \alpha < 1$).

El factor de suavizado α permite ajustar la reactividad del cálculo. Así un valor de 1 la hace equivalente a la tasa instantánea, y valores menores aumentan el peso de los cálculos anteriores. Un valor típico es $\alpha = 0{,}1$.

En Python, una implementación básica del EMA podría ser:

```
class EMA_RateMeter:
    def __init__(self, alpha=0.1):
        self.alpha = alpha
        self.rate = 0
        self.last = time.monotonic()

    def update(self, sent_bytes):
        elapsed = time.monotonic() - self.last
        self.rate = self.alpha * sent_bytes / elapsed + (1 - self.alpha) * self.rate
        self.last = time.monotonic()

ema = EMA_RateMeter()
while 1:
    data = get_data()
    sock.sendall(data)
    ema.update(len(data))
    print(ema.rate)
```

16.1.5. Consideraciones sobre el cálculo de tasa

Tampoco EMA es perfecto. Al comienzo de una conexión TCP, el emisor puede enviar —invocando `send()`— una gran cantidad de datos en muy poco tiempo porque en realidad esos datos se almacenan en el buffer de envío y recepción, pero no están siendo aún consumidos realmente por el proceso receptor.

Esto genera cálculos de tasa instantánea (en los que se basa EMA) enormes e irreales, porque a fin de cuentas lo que se está midiendo en esos primeros instantes es la tasa entre el proceso y la memoria local y remota, pero no entre los procesos emisor y receptor, que es lo que proporciona una medida representativa. Estos valores tan altos falsean el resultado, e incluso con el decaimiento exponencial, puede llevar mucho tiempo compensar ese efecto. En esta situación no mejora la reactividad, que acaba siendo tan pobre como la del promedio acumulado.

Hay algunas opciones que pueden ayudar a paliar este problema:

- Ignorar la tasa instantánea los primeros segundos de la conexión (período de calentamiento o *warmup*).
- Descartar lapsos (Δt) demasiado pequeños.
- Utilizar un α bajo para Δt pequeños. Se puede calcular como $\alpha = 1 - e^{-dt/\tau}$ siendo τ una constante que determina la reactividad.
- Reducir el tamaño de los buffers TCP de envío y recepción.
- No usar EMA en el emisor.
- Usar SMA en lugar de EMA durante el calentamiento o si no se necesita alta reactividad.

Aplicaremos algunas de estas técnicas a los ejemplos de este capítulo.

16.2. Control de flujo TCP como limitador de tasa

Sabemos que el control de flujo TCP permite al receptor controlar la cantidad de datos que el emisor le envía de acuerdo al espacio del que dispone (la ventana que anuncia). Aunque no se su objetivo principal, este mecanismo se puede utilizar para conseguir un control de la tasa de transferencia rudimentario. Veamos cómo.

Considerando solo uno de los sentidos de la comunicación, asumamos por un momento que el emisor siempre tiene datos disponibles y los puede enviar siempre que sea posible. Con esta premisa se pueden dar tres situaciones en función de la tasa de recepción (T_r) respecto a la velocidad de la red (T_n). Lógicamente la tasa de emisión (T_s) estará limitada por la menor de las otras dos.

- Si $T_r < T_n$, el buffer de recepción se llena, la ventana se cierra, el buffer de emisión también se llena y el emisor se bloquea (*stall*) en espera de que se libere espacio cuando la ventana vuelva a abrir. En esta situación, la tasa de emisión está limitada por el proceso receptor.
- Si $T_r > T_n$, el buffer de recepción queda vacío y el receptor quedará bloqueado en espera de nuevos datos. En esta situación, la tasa de emisión está limitada por la red.
- Si $T_r \approx T_n$, el buffer de recepción nunca se llena ni queda vacío. Ni el emisor ni el receptor se bloquean.

Lógicamente, los buffers de emisión y recepción existen precisamente porque el tercer caso, aunque deseable, es poco frecuente. Estas tres velocidades: emisión, red y recepción suelen ser dispares y cambiantes. Los buffers amortiguan esas diferencias y permiten un flujo más estable. Sin embargo, obviando otros controles —como el de congestión— cuando la ventana está abierta el emisor TCP envía datos a la máxima tasa posible en ese momento, y eso frecuentemente provoca bloqueos en ambos extremos, lo que influye en parte en que T_r y T_n varíen constantemente. Además el algoritmo de Nagle trata de evitar segmentos pequeños, lo que alarga esos bloqueos. La conclusión es que cuando existe disparidad en el desempeño de emisor y receptor, el mecanismo de control de flujo (y en menor medida el de congestión) provoca un flujo a ráfagas, es decir, momentos en los que se envían datos a la máxima velocidad posible y momentos en los que no se envía nada (porque la ventana está cerrada).

La consecuencia directa de todo esto es que si el receptor limita intencionadamente la tasa de recepción, la tasa media de emisión se verá también limitada, aunque su dispersión (*p. ej.* la desviación típica) será muy alta debido al flujo en ráfagas.

En principio una limitación de tasa de este tipo no es incorrecta, pero cuando ocurre de forma generalizada puede provocar efectos perjudiciales como el aumento de la latencia o el *jitter*. Lo veremos más adelante.

16.3. Limitación de tasa en el receptor

Veamos cómo implementar un control básico de tasa en el receptor. Como acabamos de ver, se puede conseguir bloqueando intencionadamente el proceso con `time.sleep()` el tiempo necesario para aproximar la tasa medida a la tasa deseada. El siguiente código calcula la tasa con el método de promedio acumulado (`ca_rate`), pero puedes cambiarlo por alguna de las otras opciones que acabamos de ver si lo prefieres.

```
1  target_rate_kBps = 200 * 1000  # 200 kB/s
2  start_time = time.monotonic()
3  received = 0
4  while 1:
5      data = sock.recv(4096)
6      if not data:
7          break
8
9      received += len(data)
10     elapsed = time.monotonic() - start_time
11     ca_rate = received / elapsed
12
13     if ca_rate <= target_rate_kBps:
14         continue
15
16     adjust_time = (received / target_rate_kBps) - elapsed
17     if adjust_time > 0:
18         time.sleep(adjust_time)
```

LISTADO 16.2: Limitación de la tasa de recepción (medida con promedio acumulado)

Hasta la **línea 14** es equivalente al Listado 16.1 salvo que aquí estamos calculando la tasa de recepción. Si este cálculo está por debajo de la tasa objetivo (`target_rate_kBps`) no hace nada, pero si está por encima, realiza un ajuste aumentando el tiempo (el denominador de la fórmula 16.2). El ajuste se obtiene como la diferencia entre el tiempo realmente transcurrido y el tiempo que corresponde a la tasa objetivo (**línea 16**).

Vamos a retomar el ejemplo del directorio /`flow-control` que ya utilizamos en el capítulo 11 para ilustrar cómo lograrlo. El emisor (cliente) envía datos al receptor (servidor) tan rápido como puede, pero el receptor limita la tasa aplicando la idea que acabamos de ver. Ambos programas calculan la tasa de envío y recepción respectivamente mediante promedio acumulado (§ 16.1.2).

Puedes probar el ejemplo con los siguientes comandos. El segundo parámetro del servidor es la tasa máxima deseada: 200 kB/s.

Cliente

```
~$ ./client.py 127.0.0.1 2000
SNDBUF: 2,626,560 bytes
 (-) sent:2,720.0 kB, CA:9176.4 kB/s
```

Servidor

```
~$ ./server.py --limit 200 2000
Client connected: ('127.0.0.1', 42244)
RCVBUF: 131,072 bytes
 received:2,724.4 kB, CA:200.1 kB/s
```

FIGURA 16.1: Limitación de la tasa en el receptor
/flow-control

Si lo ejecutas podrás ver que la transferencia en el lado del cliente se detiene durante unos segundos de vez en cuando tal como se ha explicado en la sección anterior. El receptor no para en ningún momento porque sigue consumiendo los datos que hay en el buffer de recepción. Cuando se libera suficiente espacio, la ventana se abre y el emisor envía nuevos datos durante un tiempo, hasta que el buffer se llena y la ventana se cierra de nuevo.

La Figura 16.2 es un esquema de su funcionamiento y la Figura 16.3 muestra la cantidad de datos enviados por el emisor a partir de la información obtenida de su ejecución (almacenada en el archivo `client-stats.csv`). El diagrama en escalera permite apreciar claramente los períodos de bloqueo (*stall*) debidos al cierre de la ventana.

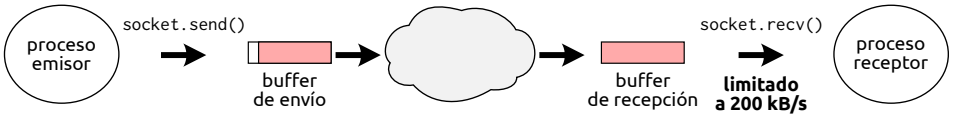

FIGURA 16.2: Limitación de tasa en el receptor

Aquí puedes apreciar perfectamente el efecto del calentamiento (*warmup*) del que hablábamos en § 16.1.5 respecto a la evolución de la tasa. Debido a que los buffers de envío y recepción son grandes: 2 624 kB y 131 kB respectivamente[4], el emisor al comienzo puede enviar muchos datos ($\sim 2,7$ MB) muy rápido. Una vez llenos los buffers, bloquea por el cierre de la ventana. Con el tiempo la tasa media se irá aproximando a los 200 kB/s impuestos por el receptor. Puedes ver esa evolución en la gráfica 16.4 obtenida también a partir de los datos generados por el emisor.

Los dientes de sierra se deben a que el emisor envía muchos datos en poco tiempo y bloquea (rampa ascendente); cuando reanuda, han pasado unos

[4]Esos son los valores por defecto en GNU/Linux para `localhost`.

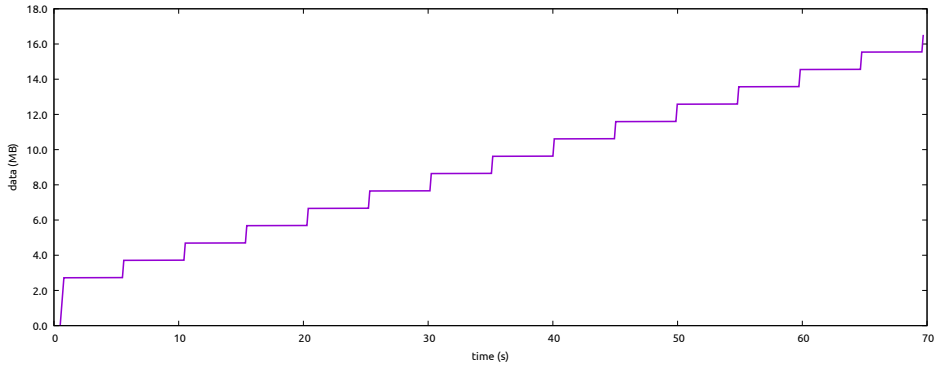

FIGURA 16.3: Datos enviados por el emisor

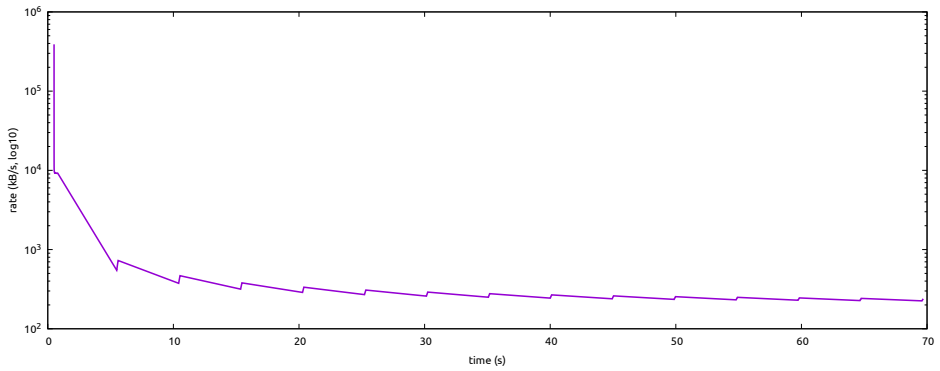

FIGURA 16.4: Tasa de envío medida en el emisor (promedio acumulado)

segundos por lo que la tasa media baja un poco (rampa descendente). Usando otro método para el cálculo de tasa, lógicamente obtendríamos gráficas sin este efecto.

Este mismo servidor proporciona alternativamente un modo diferente de trabajo (que se activa con –step). Este argumento permite especificar una cantidad de bytes, que una vez recibidos hacen que el servidor se detenga en espera de que el usuario pulse `ENTER` para continuar. De ese modo puedes comprobar fácilmente cómo el emisor bloquea al llenarse los buffers de envío y recepción y no continúa hasta que quede espacio libre en el receptor. Esta lógica está implementada en el método `step_receiving()` (Listado 16.3).

```
159     def step_receiving(self):
160         input('Press ENTER to receive > ')
161         received = 0
162         while 1:
163             count = 0
```

```
164             pending = self.step_size * 1000
165             while count < pending:
166                 data = self.conn.recv(min(4096, pending - count))
167                 if not data:
168                     return
169                 count += len(data)
170             received += count
171
172             input(f'received: {received//1000:,} kB > ')
```

LISTADO 16.3: Servidor con control manual de la recepción
/flow-control/server.py

En los ejemplos previos el cliente simplemente envía datos sin sentido, y el servidor los descarta. Sin embargo, tienen otro modo que permite indicar al cliente que consuma datos desde su entra estándar (con `--stdin`) y al servidor que envíe los datos recibidos a su salida estándar (con `--stdout`). De este modo podemos crear un sistema sencillo para transmitir un fichero `mp3` a través de la red, que será consumido en el servidor directamente por un reproductor de audio, vía redirección. Obviamente para que funcione correctamente, el cliente debe enviar un archivo adecuado (un `mp3`). Este escenario es interesante porque es el reproductor el que determina la tasa de recepción en función de la calidad y compresión del archivo (su *bitrate*). Puedes probar esa posibilidad con estos comandos:

```
~$ ./server.py --stdout --rcvbuf 4000 2000 | mpg123 -q -
~$ ./client.py --stdin --sndbuf 4000 < audio.mp3
```

FIGURA 16.5: Tasa limitada por el reproductor de `mp3`
/flow-control-player

Fíjate en las opciones `--sndbuf` y `--rcvbuf` que establecen el tamaño de los respectivos buffer de envío y recepción. Estos valores favorecen períodos de bloqueo más cortos que ayudan a entender cómo funciona la transferencia.

Tanto el cliente como el servidor muestran la tasa con promedio acumulado (CA). El servidor además puede mostrar la tasa SMA si se activa el argumento `--sma` y la tasa EMA con compensación de calentamiento basada en el ajuste de α respecto Δt si se activa el argumento `--ema`. La Figura 16.7 representa ambas tasas para comparación.

En la ejecución de la Figura 16.5 la tasa de recepción ronda los 41 kB/s (328 kbps), muy próxima al bitrate con el que está codificado el `mp3` con el que hemos realizado la prueba: 320 kbps.

```
flow-control$$ ./server.py --ema --sma --stdout --rcvbuf 4000 2000 | mpg123 -q -
Client connected: ('127.0.0.1', 52406)
RCVBUF: 8,000 bytes
 received:968.7 kB, CA:43.0 kB/s, EMA:40.8 kB/s, SMA:41.0 kB/s
```

FIGURA 16.6: Opciones para medición de tasa en el servidor: EMA y SMA
/flow-control-player/server.py

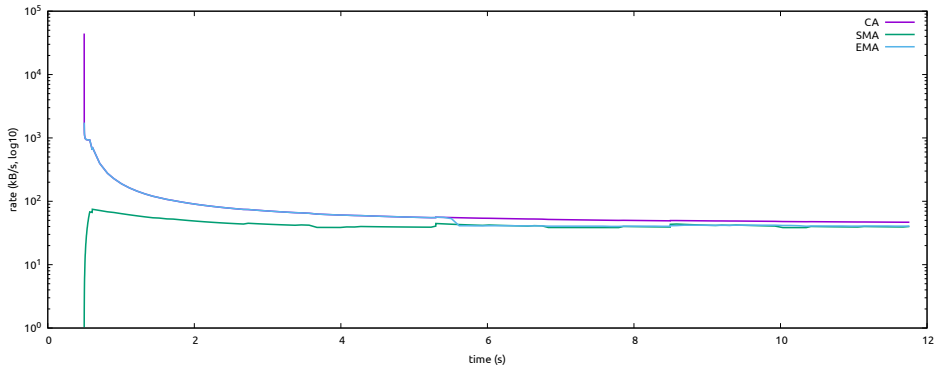

FIGURA 16.7: Comparación de CA, SMA y EMA en el servidor

La consecuencia de tener el reproductor en el receptor es la misma que en los otros ejemplos. Como la tasa de recepción es menor que la tasa de emisión, ambos buffers se llenan y el emisor queda eventualmente bloqueado de manera intermitente. En este caso, también el receptor puede quedar bloqueado porque la entrada estándar del reproductor dispone de un buffer adicional, con lo que la llamada `stdout.flush()` puede bloquear el proceso receptor hasta que haya espacio libre en ese buffer. Ten en cuenta que esto ocurre porque ambos cliente y servidor se están ejecutando en el mismo nodo y se comunican a través de `localhost`. En una conexión remota y en función del ancho de banda disponible, este comportamiento puede variar mucho, hasta el punto de no ocurrir bloqueos si ese ancho de banda es similar al bitrate al que el reproductor consume el flujo.

```
~$ file audio.mp3
audio.mp3: Audio file with ID3 version 2.4.0, contains: MPEG ADTS, layer III, v1,
320 kbps, 44.1 kHz, JntStereo
```

FIGURA 16.8: Información del `mp3` utilizado en la prueba

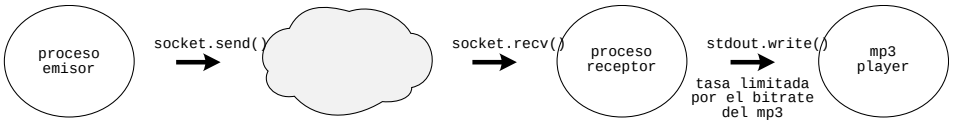

FIGURA 16.9: Datos enviados por el emisor

16.4. Limitación de tasa en el emisor

Por supuesto, el emisor puede aplicar exactamente la misma técnica intercalando pausas para conseguir una tasa de emisión concreta. Sin embargo, en este caso la tasa será mucho más estable. El emisor puede enviar bloques de datos más pequeños más a menudo, lo que reduce mucho la dispersión.

El problema es que el emisor no sabe a qué tasa consume el receptor. Si se excede, el algún momento se llenarán los buffers y se bloqueará el emisor. Si es demasiado baja, será el receptor el que se bloquee en espera de datos nuevos y la reproducción del audio se interrumpirá.

[Capítulo incompleto]