Control de congestión en TCP

Sistemas Telemáticos

Departamento de Teoría de la Señal y Comunicaciones y
Sistemas Telemáticos y Computación
URJC

Universidad Rey Juan Carlos

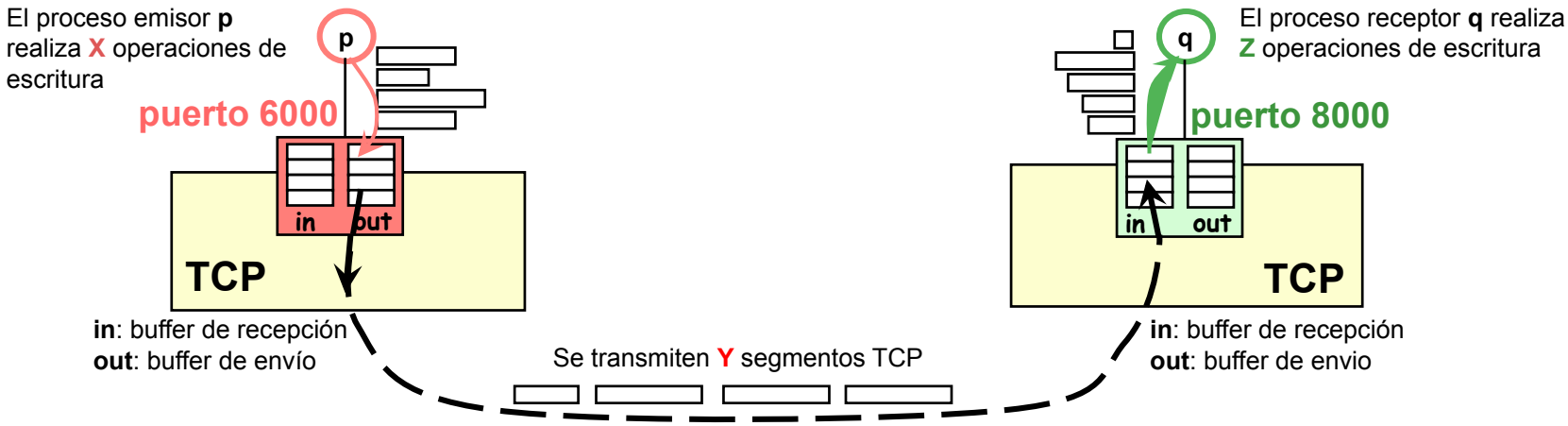
Marzo de 2018

Contenidos

Contenidos

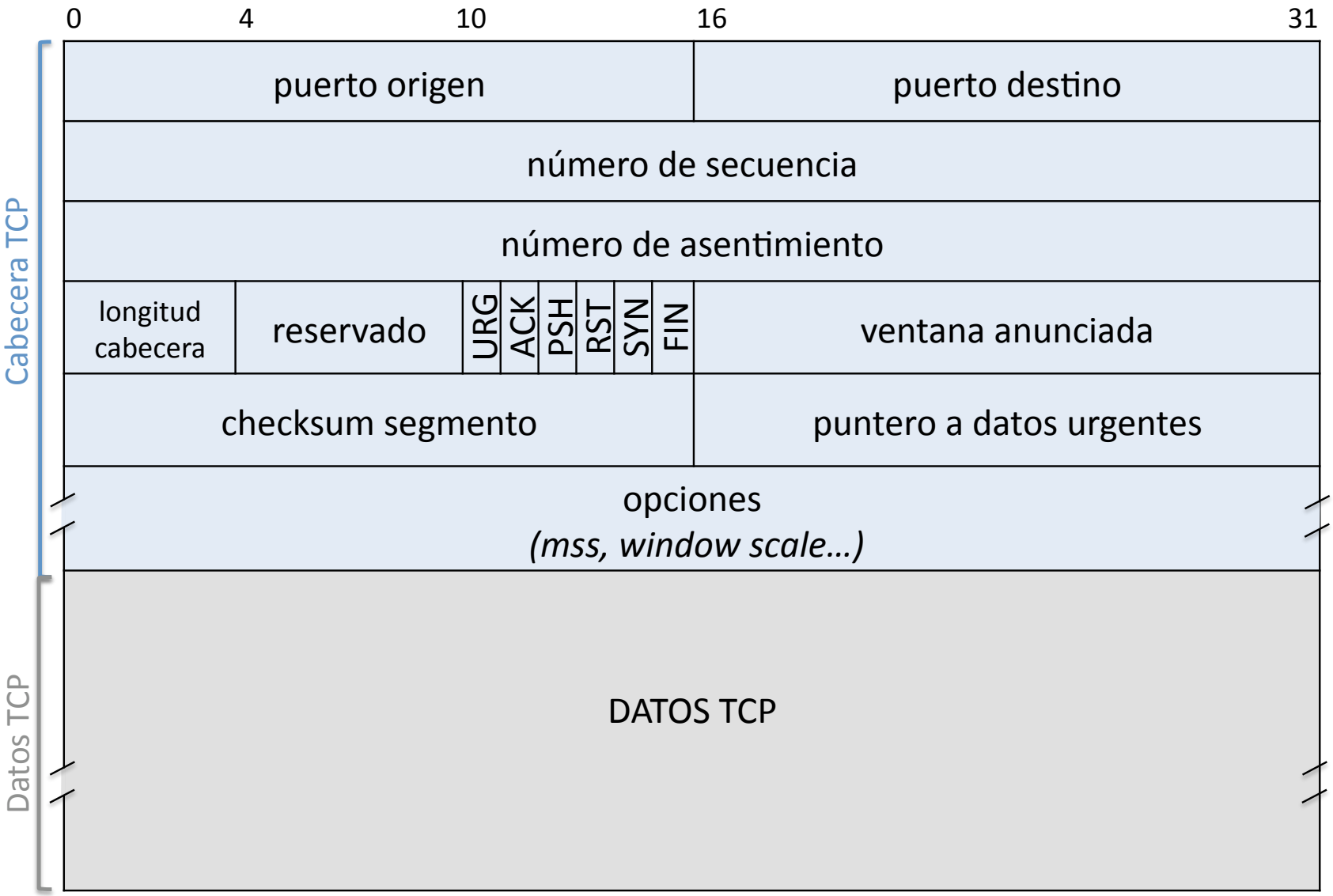
Características del protocolo TCP

- **Orientado a conexión**: Fases de establecimiento de conexión, intercambio de datos y cierre de la conexión.
- Envío de datos de forma **fiable**: el proceso receptor recibe los datos sin pérdidas, duplicados ni desorden
- Envío de datos como **flujo de bytes**:
  - El proceso emisor escribe un flujo de bytes.
  - TCP envía segmentos del tamaño que considera adecuado.
  - El proceso receptor lee un flujo de bytes.



- Las conexiones son ***full duplex***: ambos lados pueden enviar datos simultaneamente.

Formato del segmento TCP



Números de secuencia y números de ack

- Se numeran los bytes, no los segmentos.
- Al establecerse una conexión se parte de un número de secuencia inicial. Cada lado de la conexión usa sus propios números de secuencia, independientes de los que usa el otro lado.
  - `wireshark` por defecto muestra números de secuencia relativos al principio de la conexión
- El número de ack indica **el siguiente byte de datos que espera recibir** quien envía el segmento. Podría tener bytes posteriores, pero el número de ack corresponde al primer byte que le falta desde el principio de la conexión.
  - Una opción de TCP permite usar acks selectivos (SACKs).

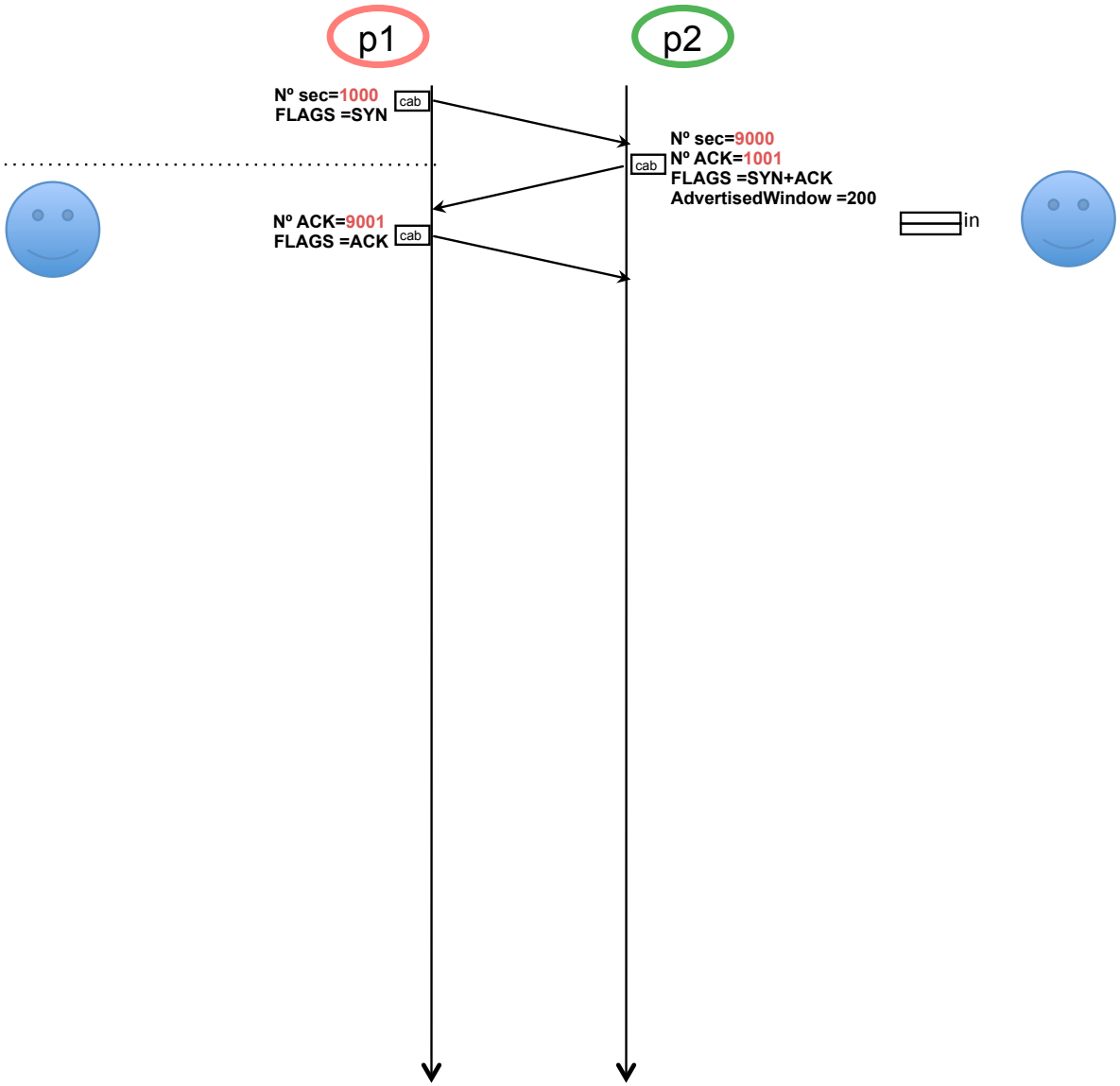
Retransmisiones y plazo de retransmisión

- Cuando se envía un segmento se arranca un temporizador con un **plazo de retransmisión** para esperar su asentimiento.
- ***Timeout***: Se produce cuando expira el plazo de retransmisión de un segmento sin que se haya recibido aún un ACK que lo asienta. Al producirse un *timeout*, se retransmite el segmento.
- El valor del plazo de retransmisión se estima a partir de los RTT medidos para cada segmento asentido, y es aproximadamente igual al doble del RTT.
  - RTT *(round-trip time)*: tiempo desde que se envía un segmento hasta recibe su asentimiento
- ***Exponential backoff***: Cada vez que se produce un nuevo *timeout* sobre el mismo segmento (y por lo tanto se retransmite), **se dobla el plazo de retransmisión**.

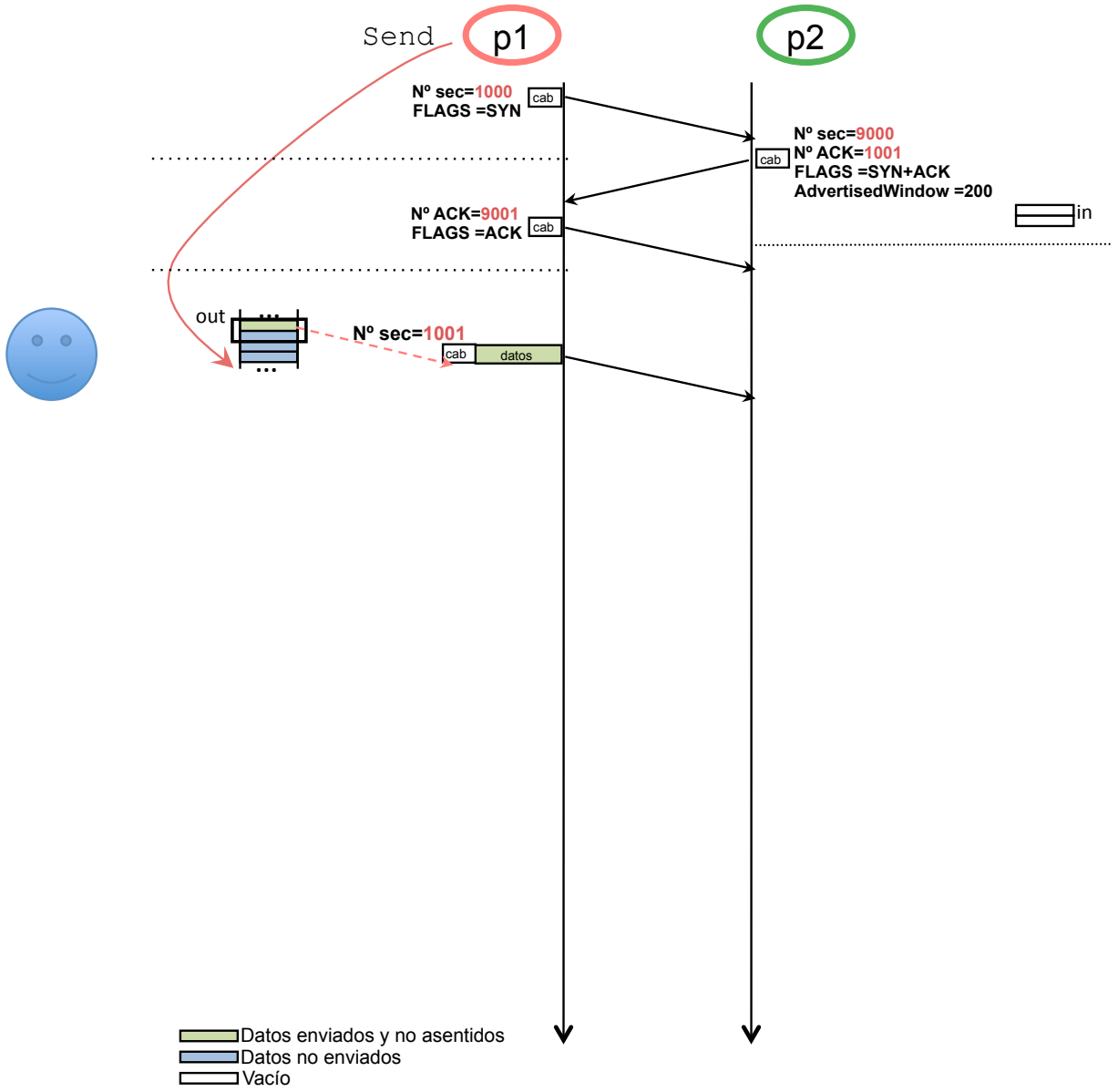
Ventana Anunciada (también llamada Ventana de Flujo)

- El receptor anuncia al emisor en el campo de **Ventana Anunciada** (`AdvertisedWindow`) el número de bytes que acepta a partir del **Número de ACK**.
  - Ejemplo: Un segmento con `ACK= 1001`, `Win= 300` indica que quien lo envía acepta recibir 300 bytes a partir del 1001, es decir, acepta los números de secuencia `1001` a `1300`.
- Si el emisor transmite todo lo que le permite la ventana anunciada sin recibir ningún ACK, debe parar de transmitir datos, hasta que:
  - reciba un segmento que aumente el número de ACK, o bien
  - reciba un segmento que aumente el tamaño de la ventana anunciada
- Como una conexión es *full duplex*, cada sentido de la comunicación tiene sus valores independientes de número de secuencia y tamaño de ventana anunciada.

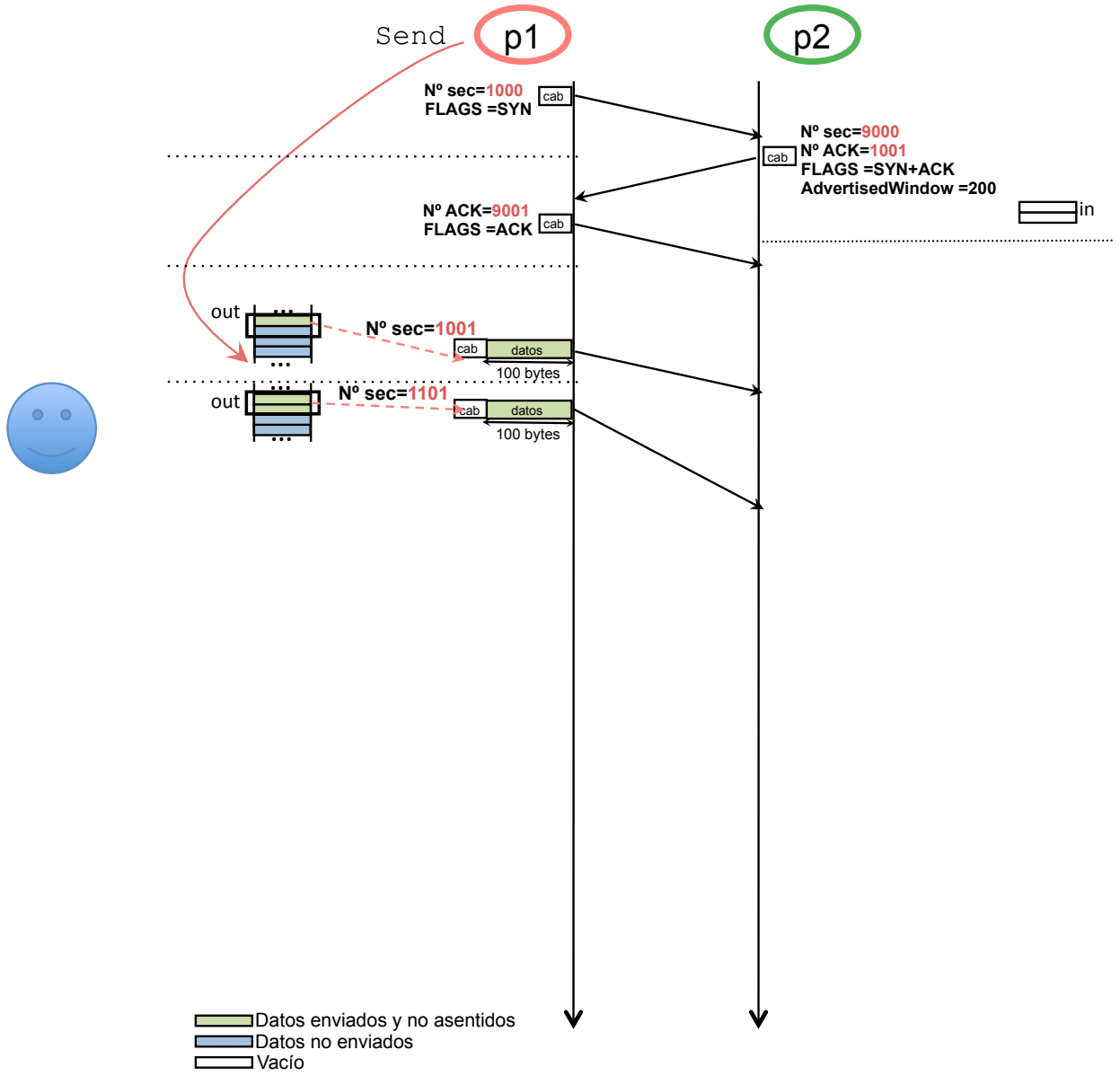
Ejemplo


p1
p2
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
cab
Nº ACK=9001
FLAGS =ACK
cab
in

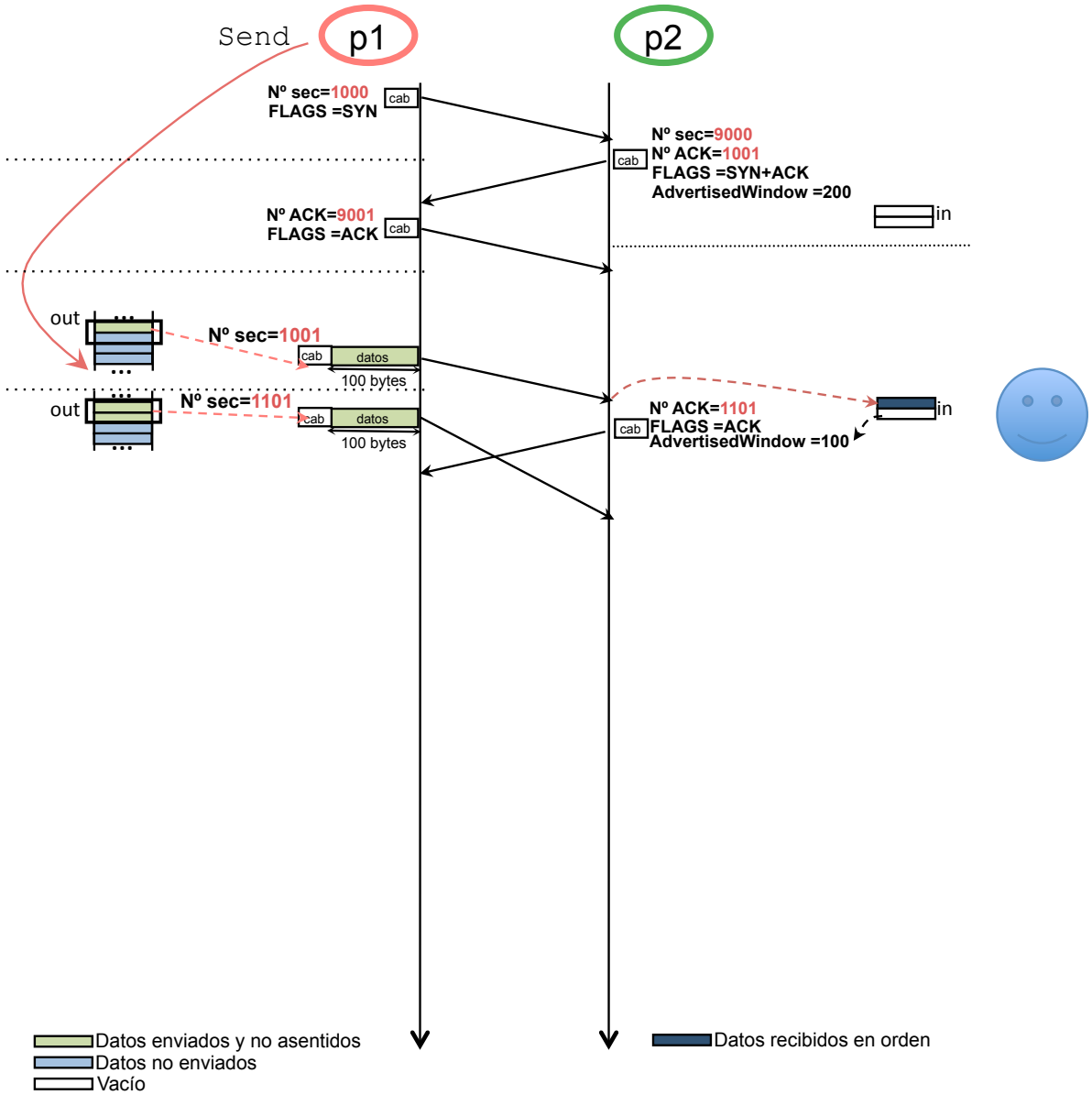
Ejemplo


Send
p1
p2
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
Datos enviados y no asentidos
Datos no enviados
Vacío

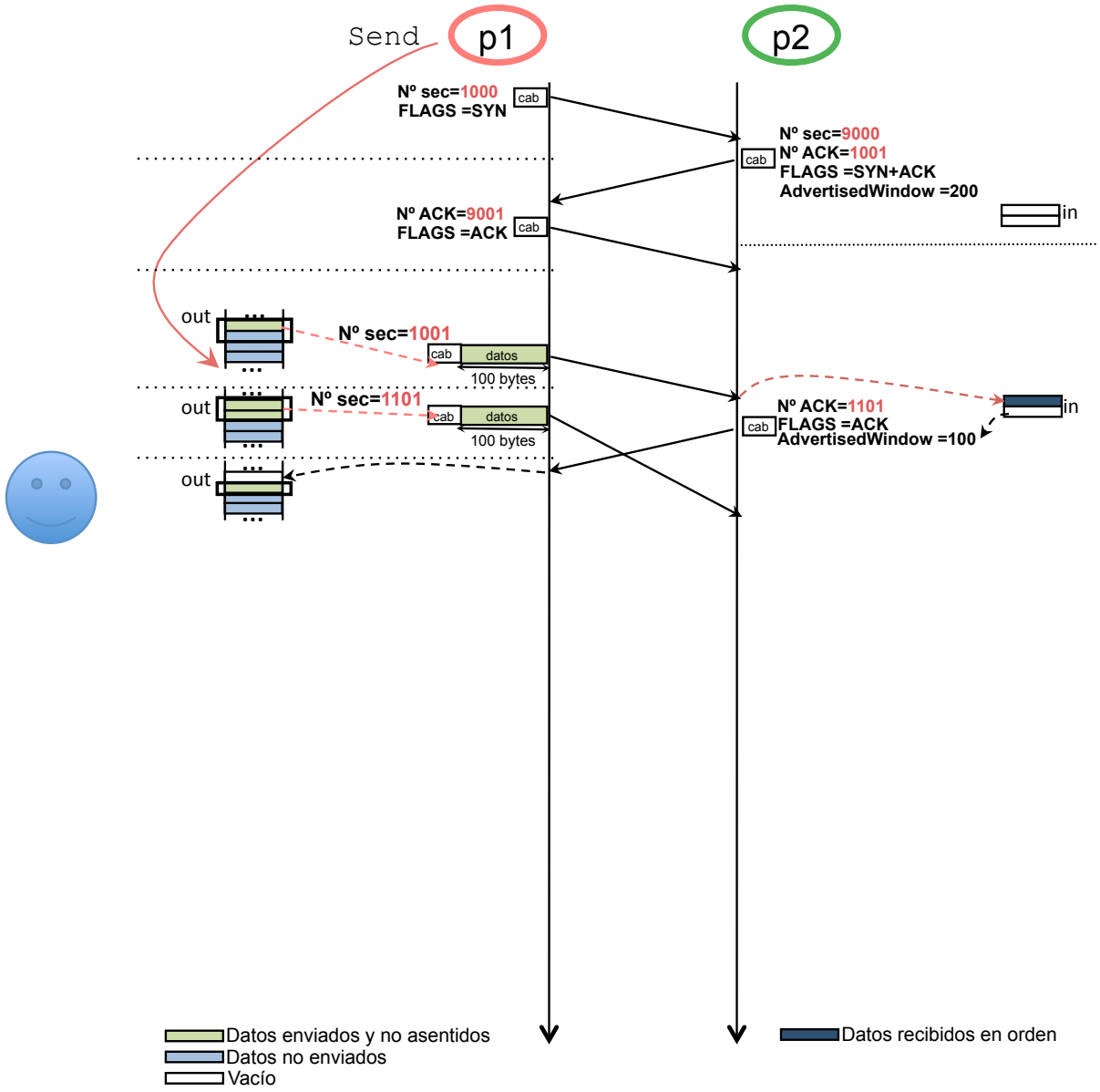
Ejemplo


Send
p1
p2
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
out
Nº sec=1101
datos
100 bytes
Datos enviados y no asentidos
Datos no enviados
Vacío

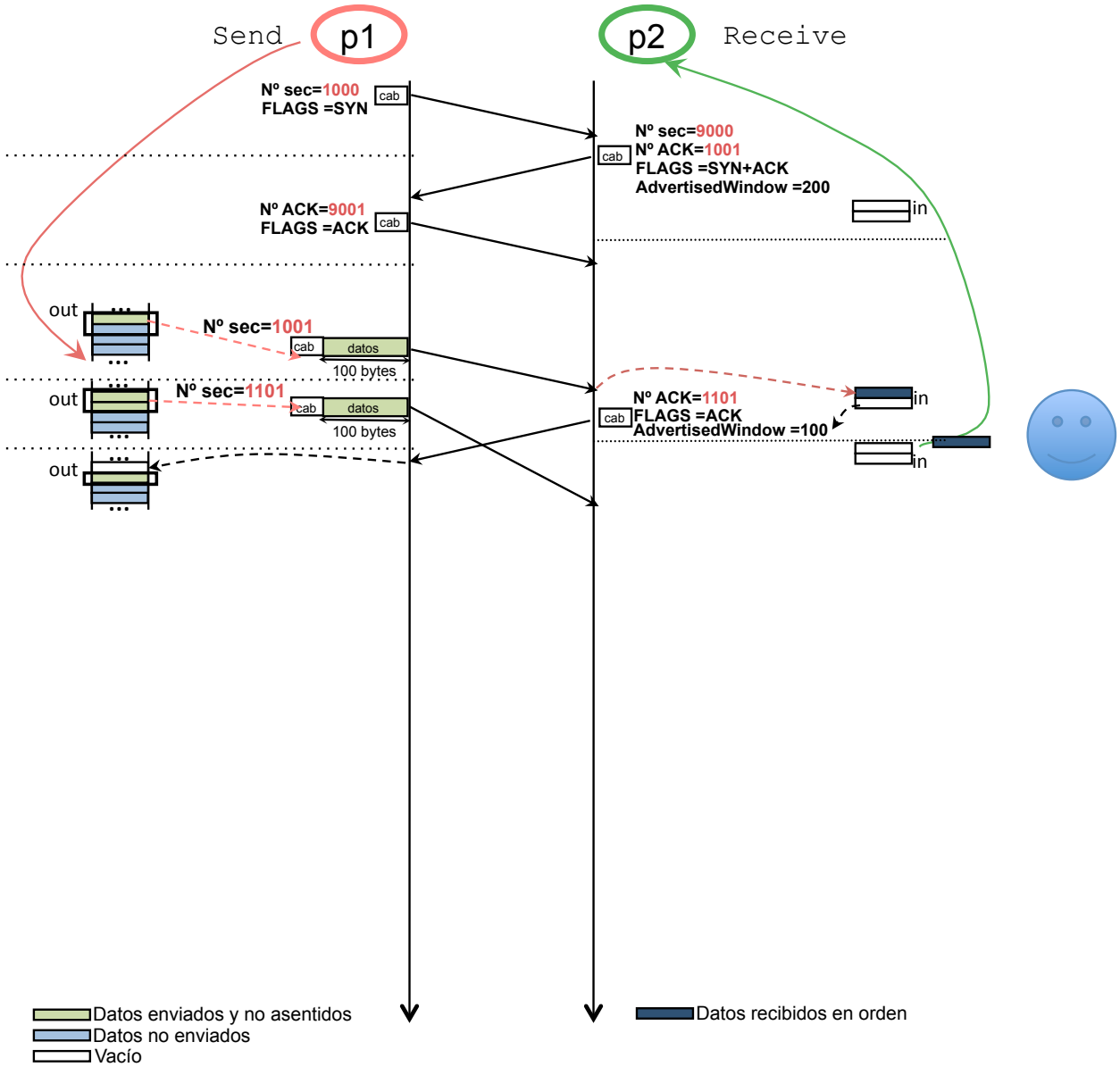
Ejemplo


Send
p1
p2
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
Nº ACK=9001
FLAGS =ACK
in
out
Nº sec=1001
datos
100 bytes
out
Nº sec=1101
datos
100 bytes
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

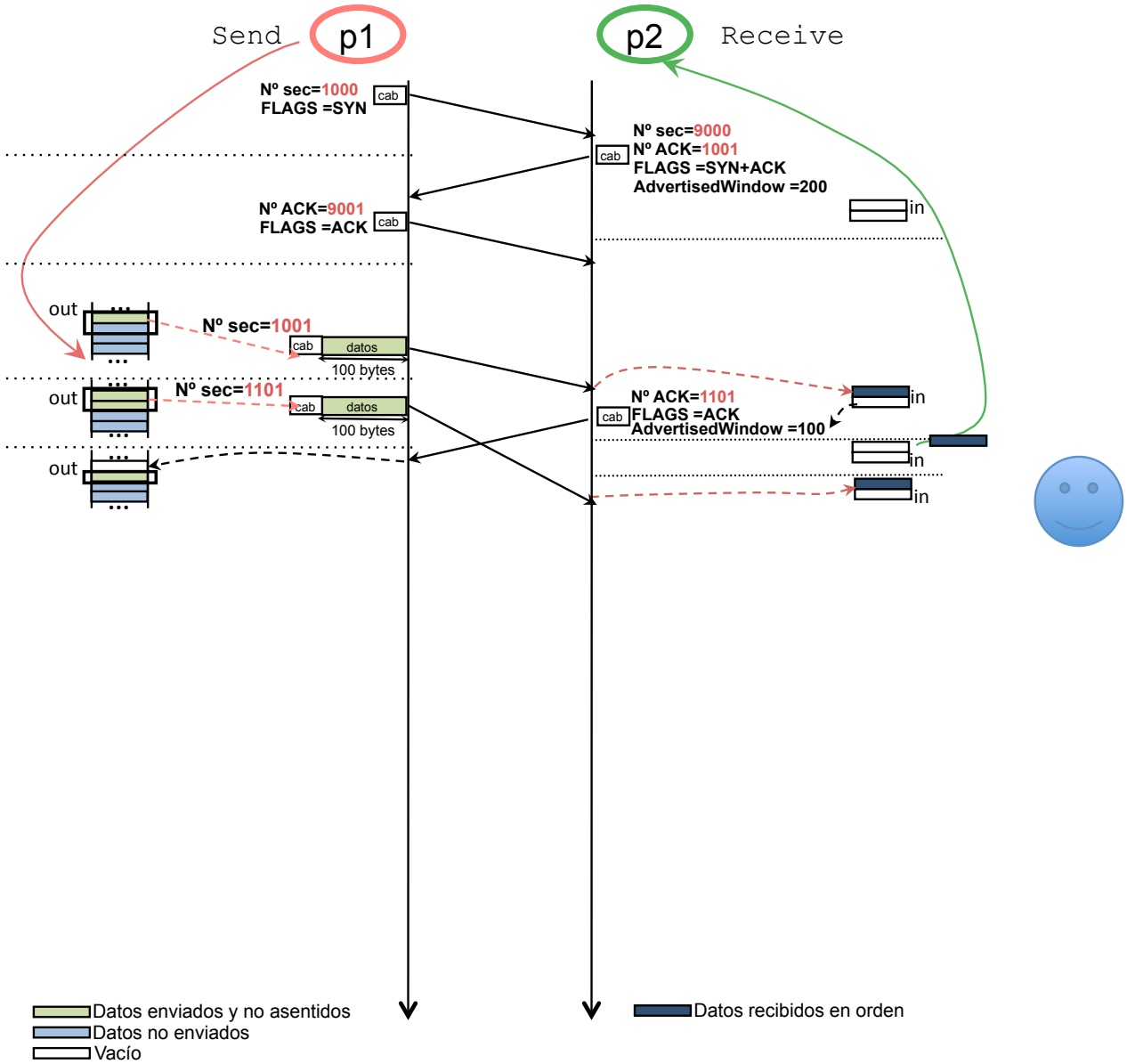
Ejemplo


Send
p1
p2
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
Nº ACK=9001
FLAGS =ACK
in
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

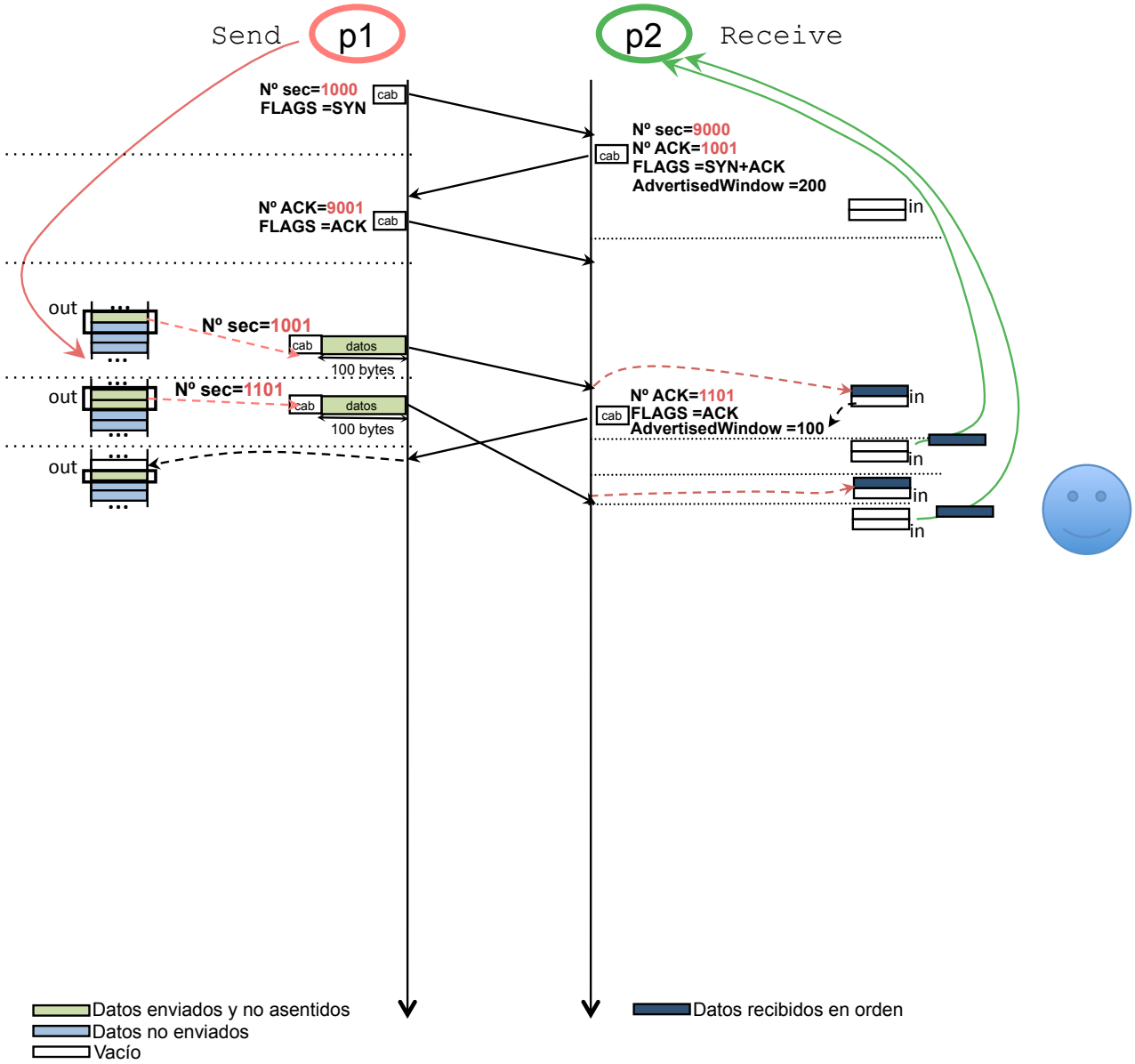
Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

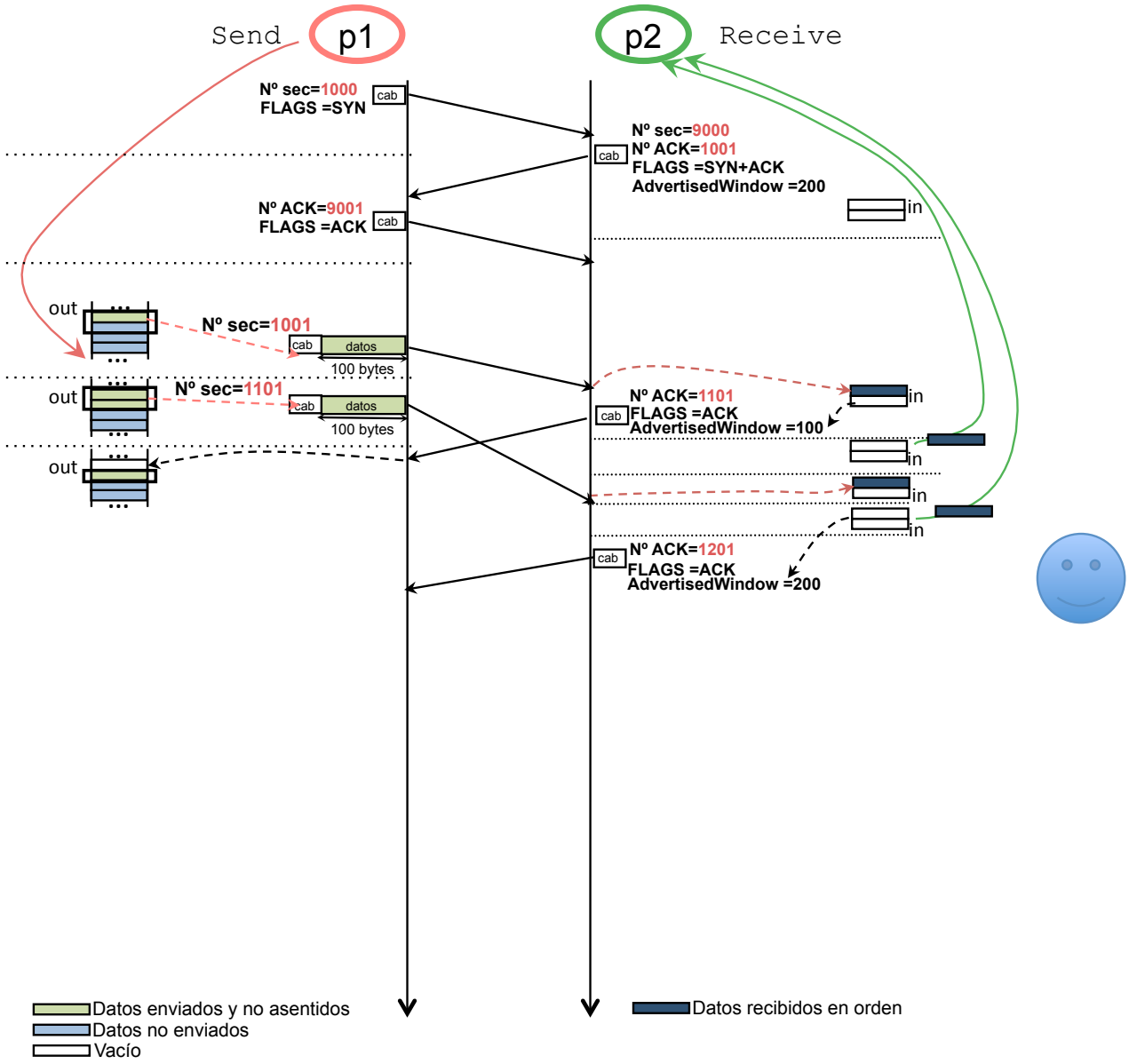
Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

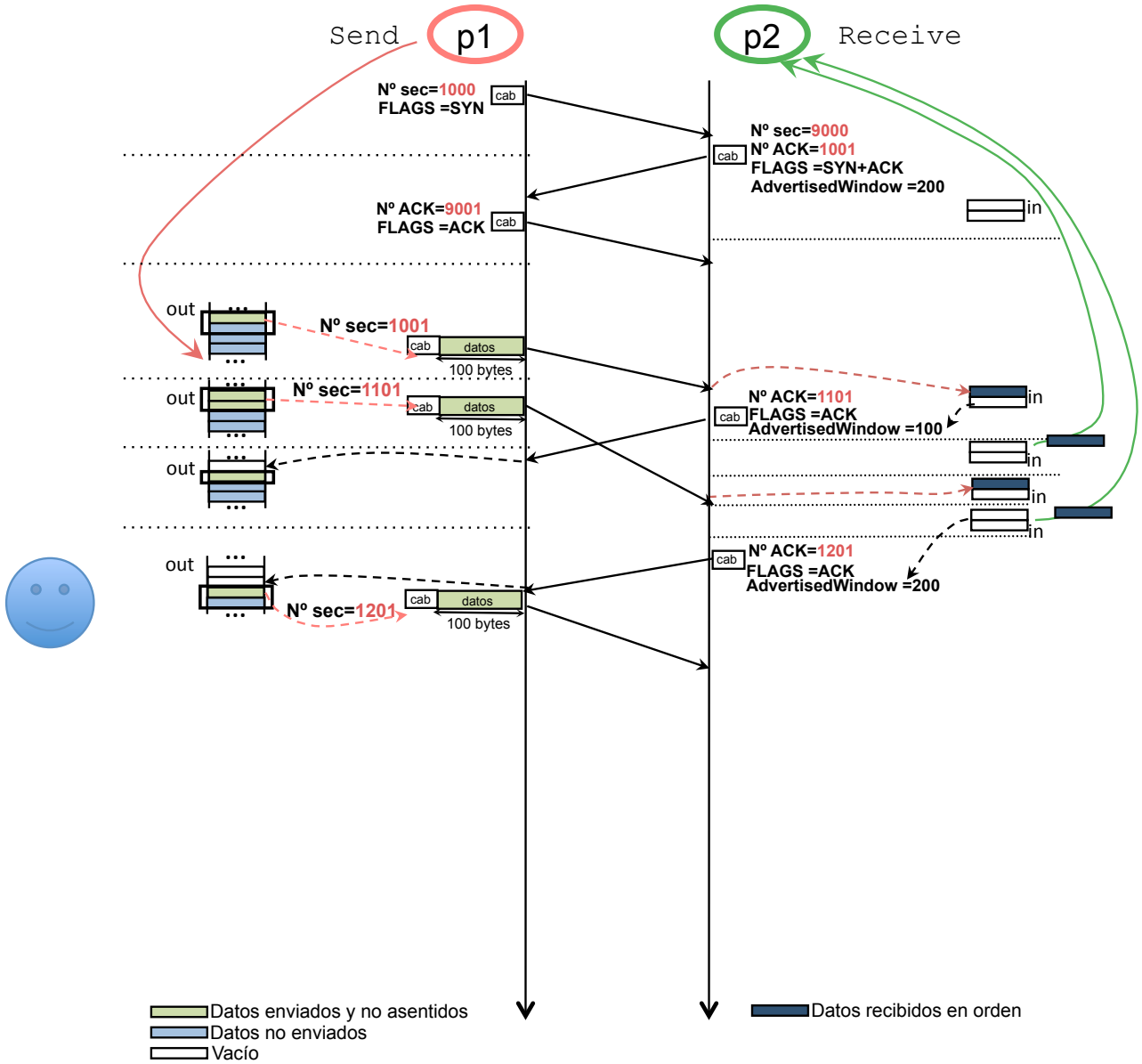
Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
in
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

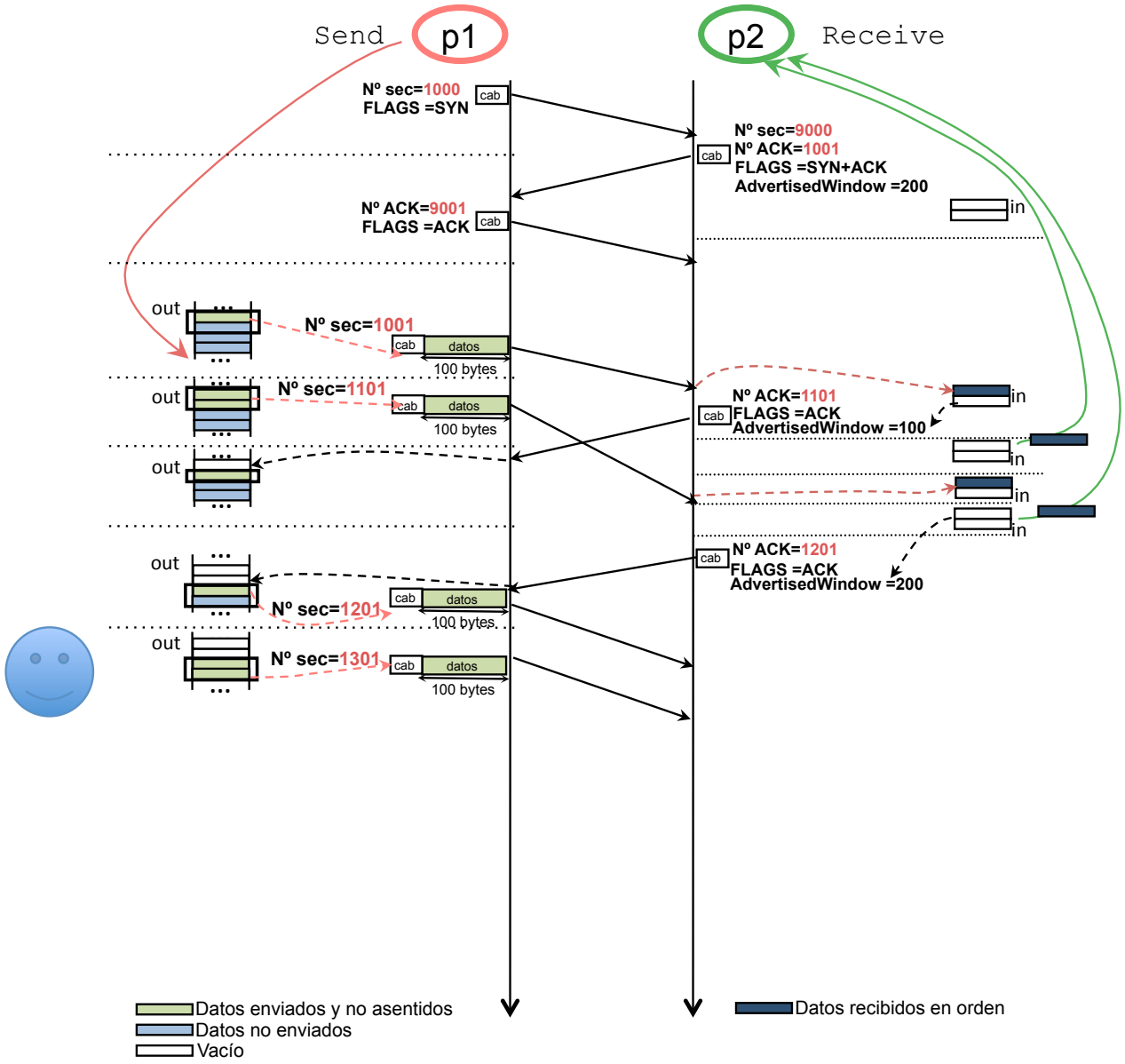
Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1201
FLAGS =ACK
AdvertisedWindow =200
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

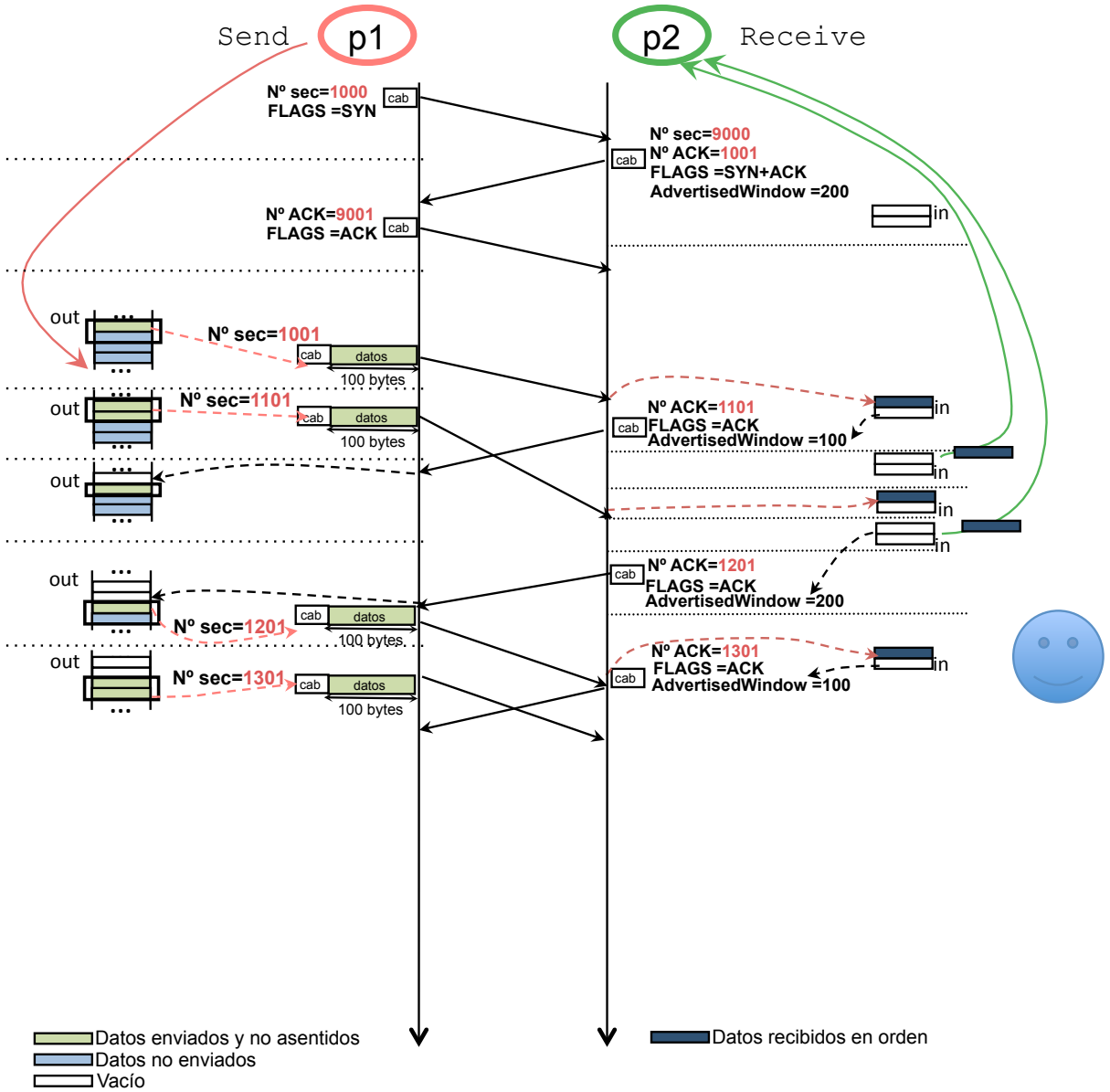
Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1201
FLAGS =ACK
AdvertisedWindow =200
Nº sec=1201
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

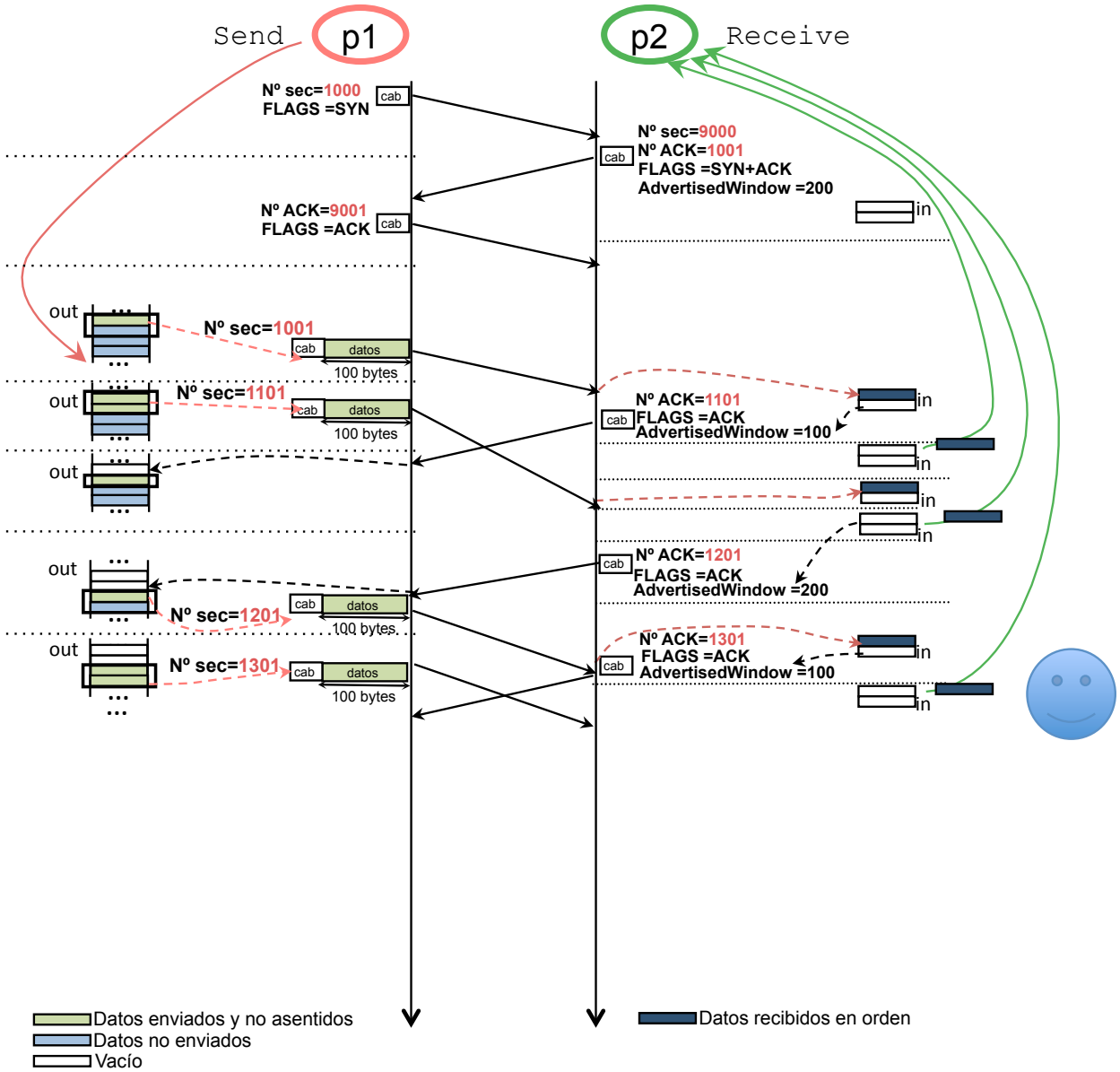
Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
Nº ACK=9001
FLAGS =ACK
in
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1201
FLAGS =ACK
AdvertisedWindow =200
Nº sec=1201
Nº sec=1301
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

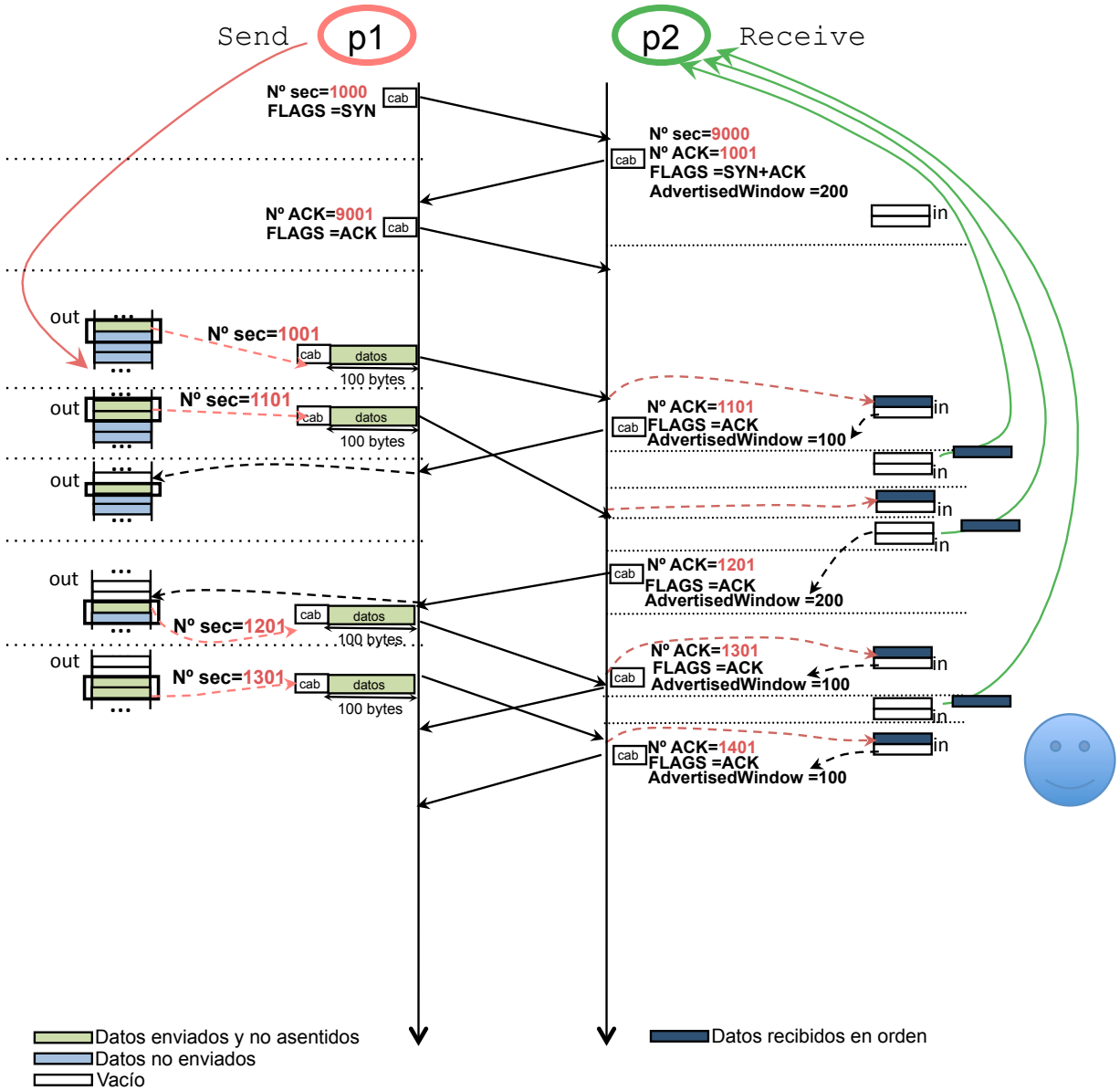
Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1201
FLAGS =ACK
AdvertisedWindow =200
Nº sec=1201
Nº sec=1301
Nº ACK=1301
FLAGS =ACK
AdvertisedWindow =100
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

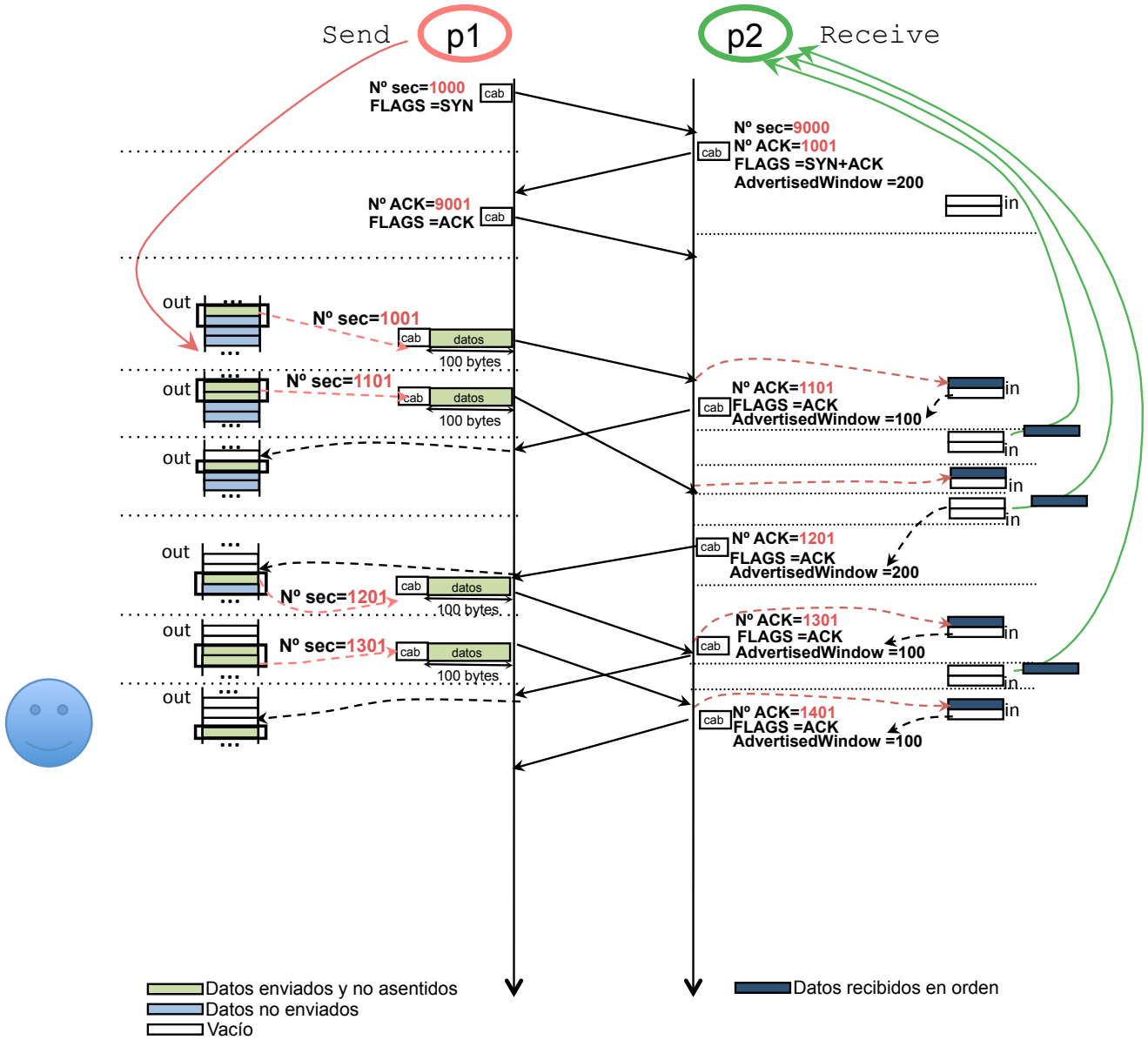
Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1201
FLAGS =ACK
AdvertisedWindow =200
Nº sec=1201
Nº ACK=1301
FLAGS =ACK
AdvertisedWindow =100
Nº sec=1301
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

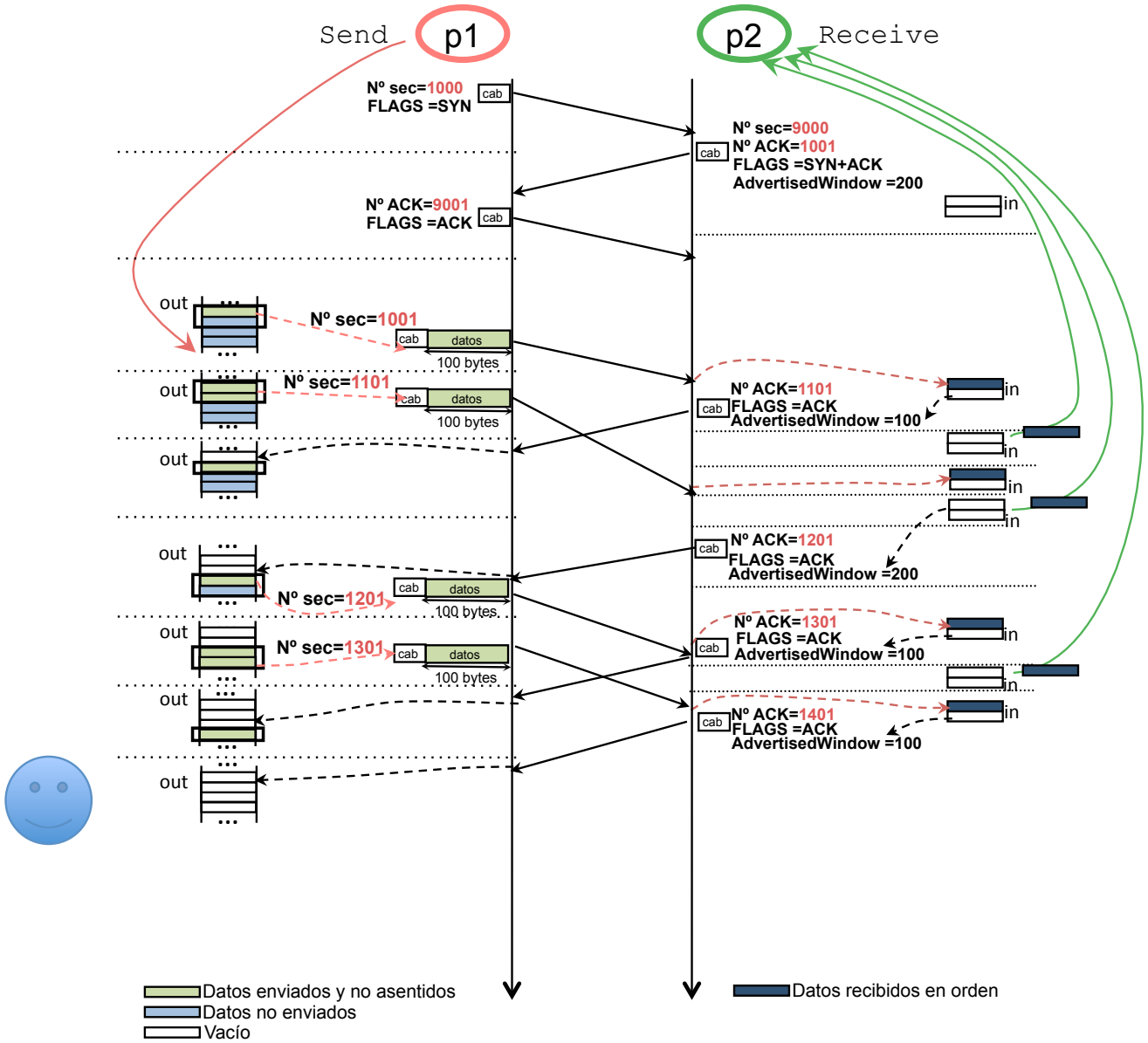
Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1201
FLAGS =ACK
AdvertisedWindow =200
Nº sec=1201
Nº sec=1301
Nº ACK=1301
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1401
FLAGS =ACK
AdvertisedWindow =100
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

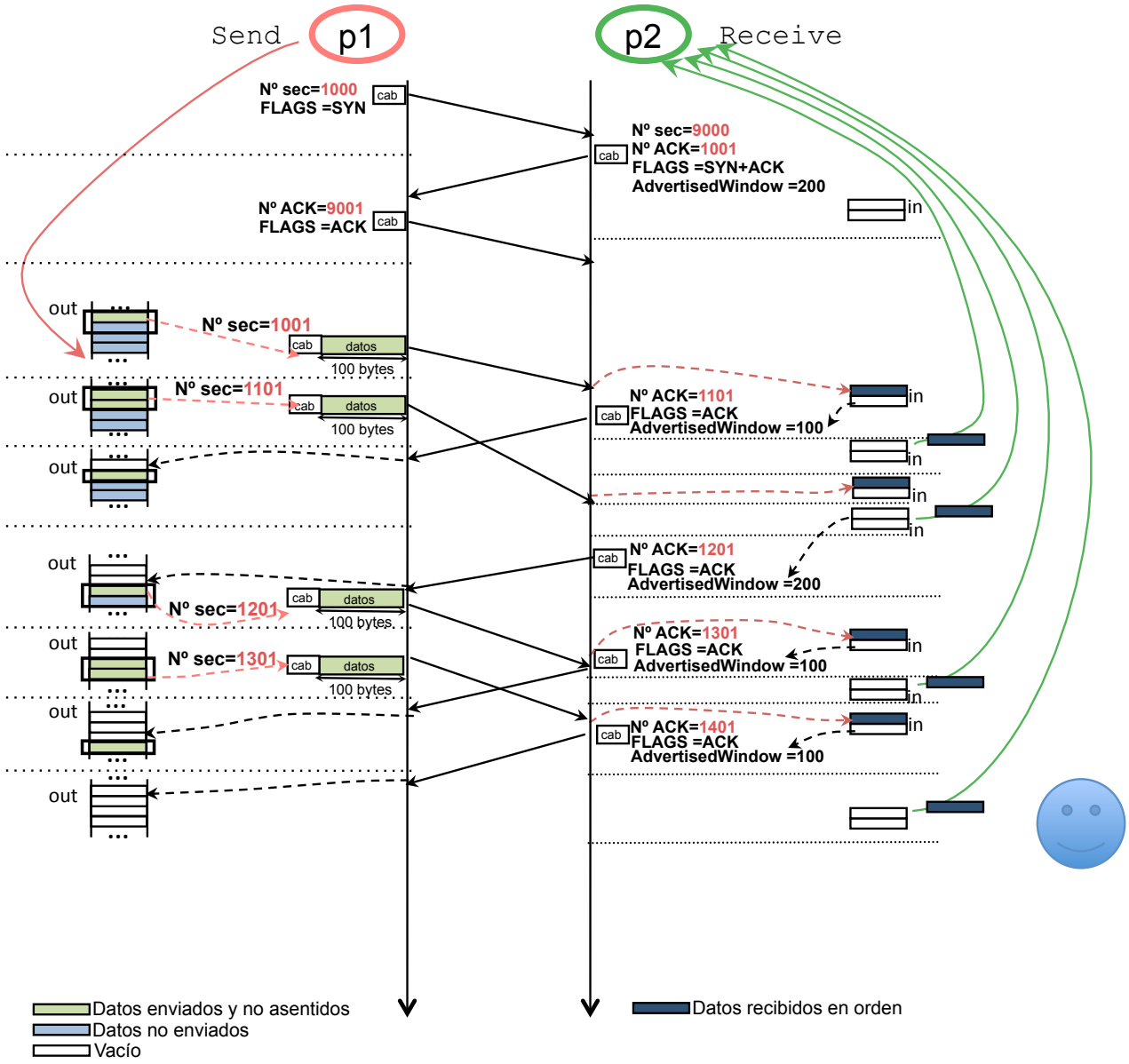
Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1201
FLAGS =ACK
AdvertisedWindow =200
Nº sec=1201
Nº sec=1301
Nº ACK=1301
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1401
FLAGS =ACK
AdvertisedWindow =100
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
in
Nº ACK=1201
FLAGS =ACK
AdvertisedWindow =200
Nº sec=1201
Nº sec=1301
Nº ACK=1301
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1401
FLAGS =ACK
AdvertisedWindow =100
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

Ejemplo


Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1201
FLAGS =ACK
AdvertisedWindow =200
Nº sec=1201
Nº sec=1301
Nº ACK=1301
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1401
FLAGS =ACK
AdvertisedWindow =100
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

Ejemplo

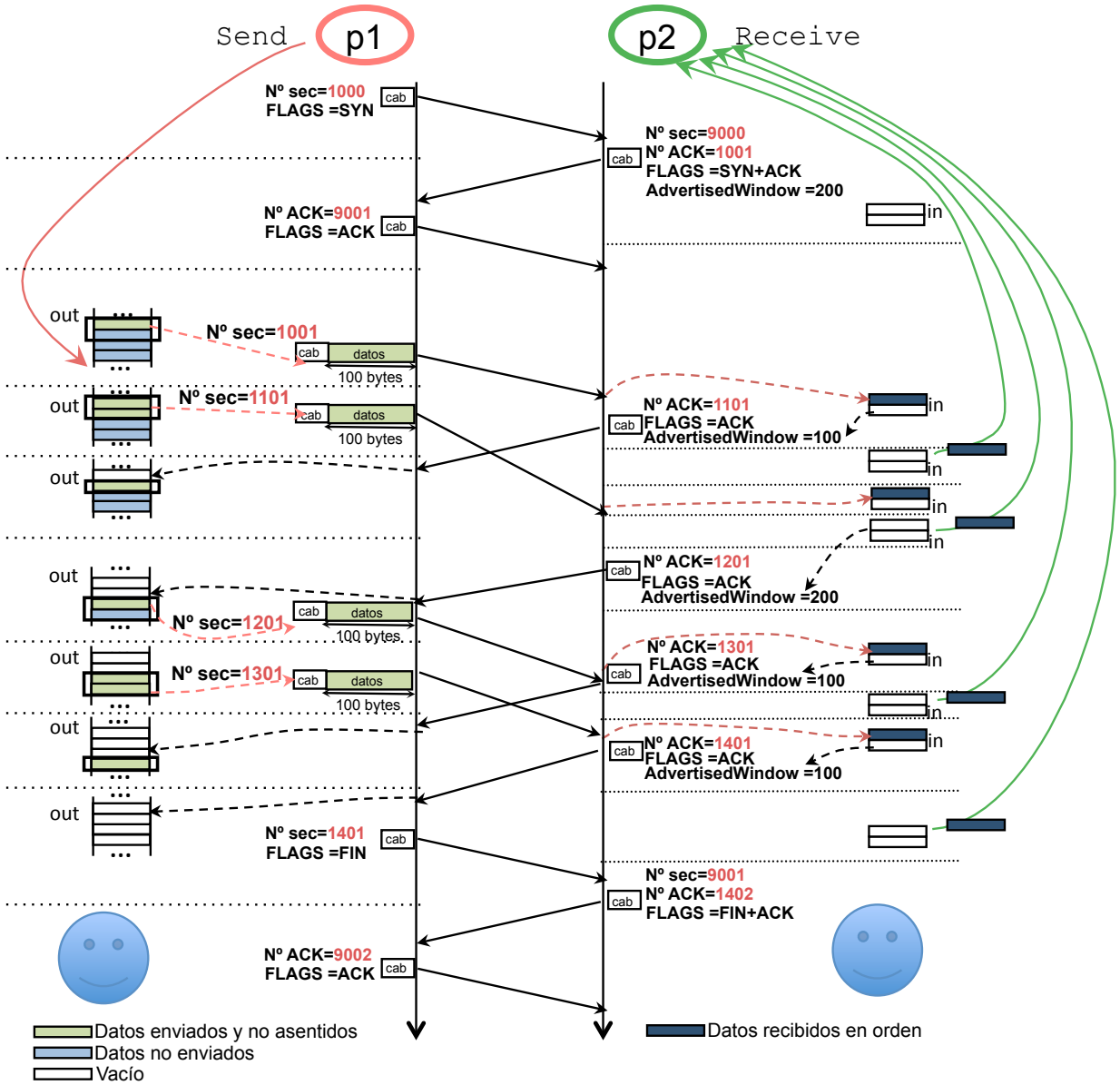
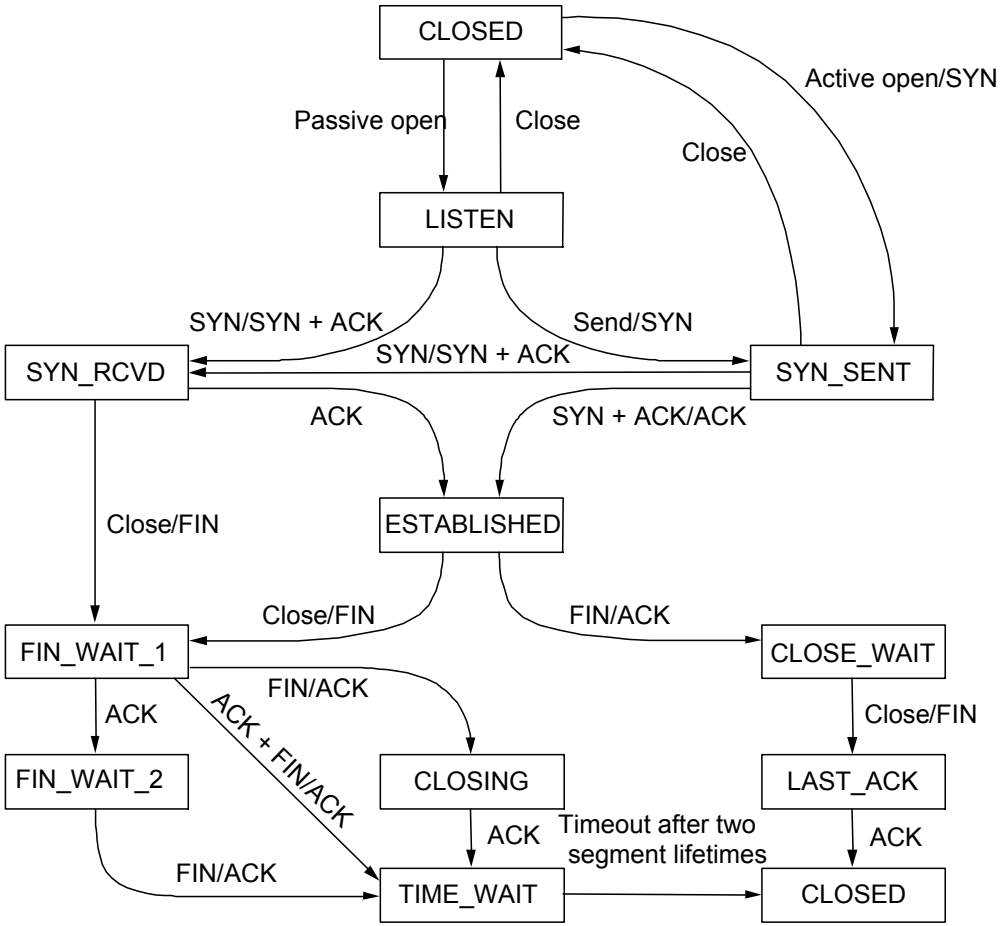

Send
p1
p2
Receive
Nº sec=1000
FLAGS =SYN
cab
Nº sec=9000
Nº ACK=1001
FLAGS =SYN+ACK
AdvertisedWindow =200
in
Nº ACK=9001
FLAGS =ACK
out
Nº sec=1001
datos
100 bytes
Nº sec=1101
Nº ACK=1101
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1201
FLAGS =ACK
AdvertisedWindow =200
Nº sec=1201
Nº sec=1301
Nº ACK=1301
FLAGS =ACK
AdvertisedWindow =100
Nº ACK=1401
FLAGS =ACK
AdvertisedWindow =100
Nº sec=1401
FLAGS =FIN
Nº sec=9001
Nº ACK=1402
FLAGS =FIN+ACK
Nº ACK=9002
FLAGS =ACK
Datos enviados y no asentidos
Datos no enviados
Vacío
Datos recibidos en orden

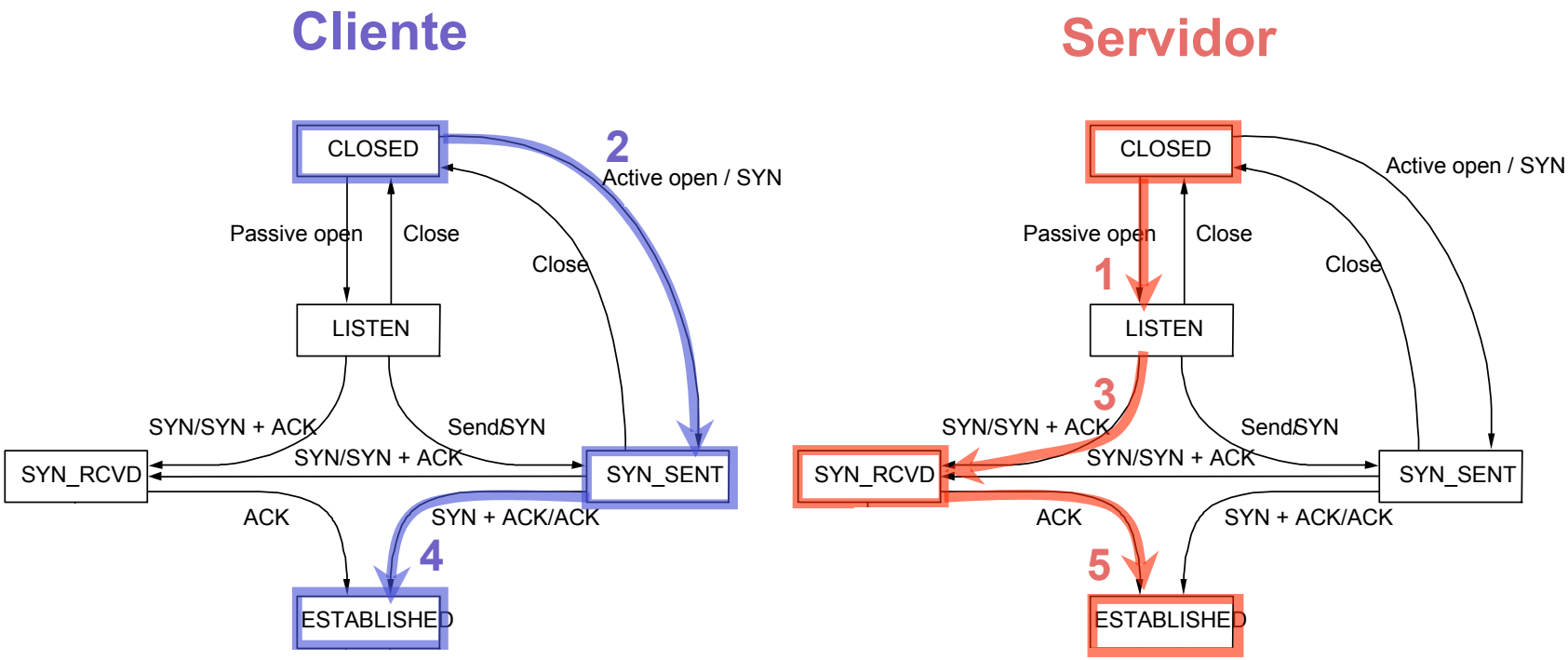
Diagrama de estados en el establecimiento y cierre de la conexión TCP



X/Y:

- X: Operación de la aplicación o segmento recibido que provoca el cambio de estado
- Y: Segmento enviado

Diagrama de estados en el establecimiento de la conexión TCP

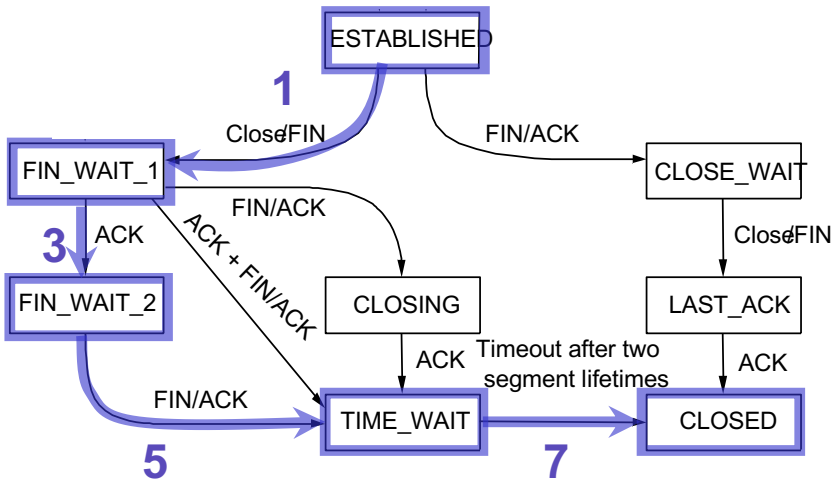


X/Y:

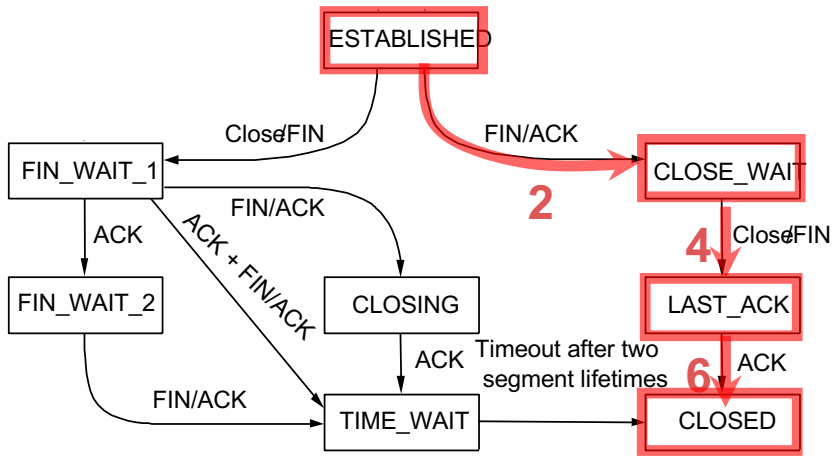
- X: Operación de la aplicación o segmento recibido que provoca el cambio de estado
- Y: Segmento enviado

Diagrama de estados en el cierre de la conexión TCP

Lado que envía el primer FIN



Lado que recibe el primer FIN



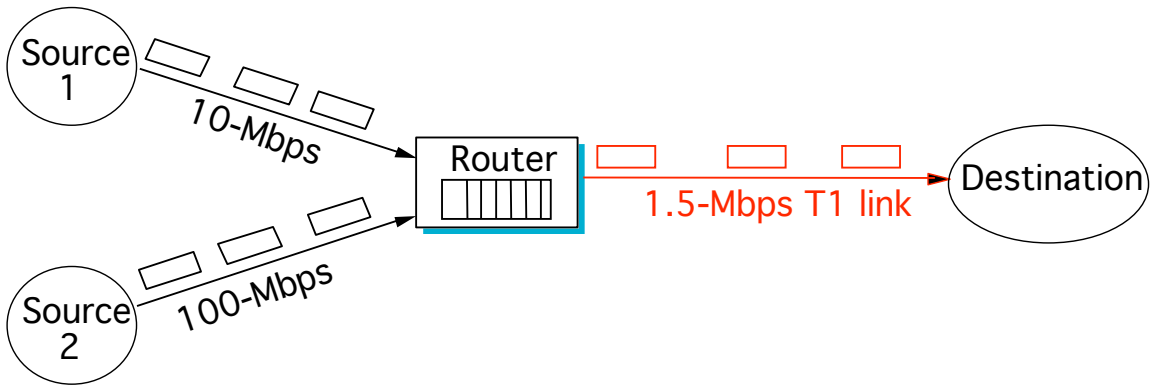
X/Y:

- X: Operación de la aplicación o segmento recibido que provoca el cambio de estado
- Y: Segmento enviado

Contenidos

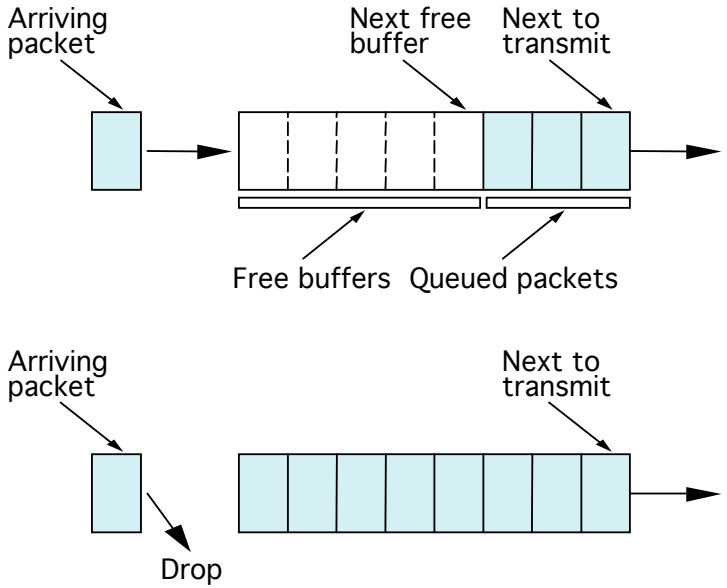
Control de flujo y control de congestión

- **Control de flujo**: Que **el receptor** no sea desbordado por un emisor que transmite más rápido de lo que él es capaz de procesar.
  - Extremo a extremo.
  - El receptor limita la velocidad de transferencia del emisor debido fundamentalmente al tamaño del *buffer* de recepción y a la velocidad con la que lee la aplicación receptora
- **Control de congestión**: Que **un router intermedio** no sea desbordado por un emisor que transmite más rápido de lo que él es capaz de procesar.
  - Involucra a extremos y a la red (encaminadores, enlaces)
  - La red limita la velocidad de transferencia del emisor



Control de congestión en TCP

- Supone red *best-effort*, encaminadores FIFO, *tail drop*
  - Red *best-effort*: se pierden paquetes y el nivel de red no recupera dichas perdidas.
  - En los *routers*, la elección del siguiente paquete para transmitir es FIFO.
  - Si se llenan los *buffers* en el encaminador, los siguientes paquetes que lleguen se descartan *(tail drop)*, independientemente de su origen.

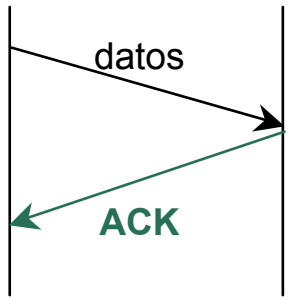


Control de congestión en TCP

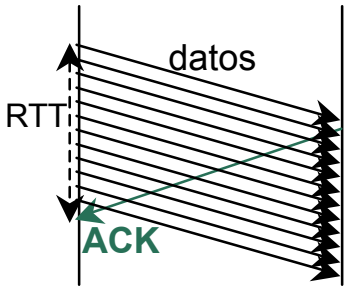
- Cada emisor determina la capacidad de la red
  - Basándose en observaciones de la transmisión: ¿le llegan todos los ACK? ¿Con qué RTT?
  - Objetivo: tener lleno el canal (aprovechar al máximo la capacidad de la red). **Maximizar BDP (Bandwidth Delay Product)**. Para ello, se envían tantos datos paquetes como:

    `RTT x bandwidth_entre_Emisor_Receptor`

- Los ACK regulan la transmisión (*self-clocking*): si llega un ACK es porque un segmento ha abandonado la red, luego hay hueco para enviar un nuevo segmento.
- Dificultades a resolver:
  - Determinar la capacidad de la red al empezar
  - Ajustarse después a los cambios en la capacidad



Infrautilización de la red



Máximo aprovechamiento de la capacidad de la red

Ventana de control de congestión

- Objetivos:
  - Determinar la capacidad de la red al iniciar una conexión
  - Ajustarse a los cambios en la capacidad disponible
- Se utiliza una nueva variable para cada sentido de la transmisión de una conexión, la **ventana de control de congestión**: `cwnd`
- La ventana de control de congestión limita cuántos datos tiene en tránsito un origen.
- Ahora el emisor tendrá en cuenta tanto la ventana anunciada (`AdvertisedWindow`) como la ventana de control de congestión (`cwnd`), y no podrá superar ninguna de las dos. Luego:

```
MaxWin = MIN(cwnd, Ventana_Anunciada)
EffWin = MaxWin - (Últ_Byte_Enviado - Últ_Byte_Asentido)
```

- Idea:
  - disminuir `cwnd` cuando aumente la congestión
  - incrementar `cwnd` cuando disminuya la congestión

Detección de congestión

- ¿Cómo determina el origen si la red está congestionada o no?

  Cuando vencen *timeouts* y hay que retransmitir:

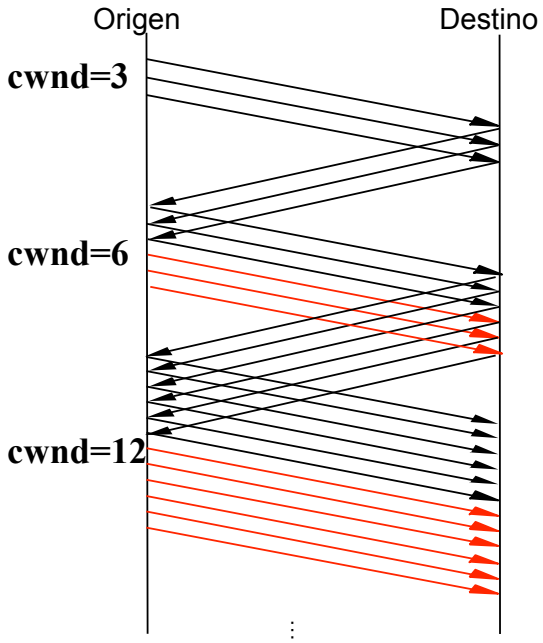
  - Se considera que un *timeout* significa que un paquete se ha perdido.
    - No tendría por qué ser así: puede haberse retrasado el ACK
  - Se supone que los paquetes casi siempre se pierden por congestión en los *routers* y que rara vez se pierden debido a errores de transmisión
    - Si la suposición no es cierta, como ocurre en redes inalámbricas, el control de congestión de TCP no funcionará adecuadamente.
- Por lo tanto, **cuando vence un *timeout* TCP supone que hay congestión**.
- Algoritmos para el control de la congestión (RFC5681):
  - *Slow Start*
  - *Congestion Avoidance*
  - *Fast Retransmit*
  - *Fast Recovery*

Contenidos

Contenidos

Slow Start en el inicio de la conexión

- ***Slow Start* (SS)** se utiliza para descubrir **rápidamente** la capacidad de la red.
- SS se aplica en el **inicio de una conexión**:
  1. Se comienza con una ventana de control de congestión:
     - Si MSS >2190 bytes => `cwnd=2*MSS` y no debe ser mayor de **2 segmentos**
     - Si MSS >1095 y MSS <= 2190 bytes => `cwnd=3*MSS` y no debe ser mayor de **3 segmentos**
     - Si MSS <=1095 bytes => `cwnd=4*MSS` y no debe ser mayor de **4 segmentos**
  2. Se dobla `cwnd` cuando se reciben los ACKs de todos los paquetes enviados según `cwnd` (aprox. cada RTT):
     - Dicho de otra forma: la `cwnd` se incrementa en 1 MSS cada vez que se recibe un ACK



OBSERVA: En SS cada ACK recibido permite enviar 2 nuevos segmentos, uno que ocupa el sitio que deja el paquete asentido, y otro por el aumento de cwnd en 1 MSS.

Slow Start cuando hay timeout, cwnd=1

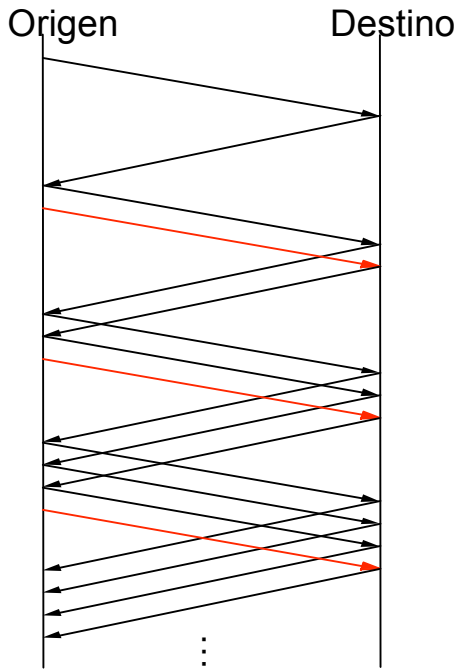
- SS se aplica también **cuando hay *timeout***:
  Pasa demasiado tiempo sin recibir ACKs → TCP considera que hay una pérdida de paquete → se ha sobrepasado la capacidad actual de la red. Por ello:
  1. Se calcula `ssthresh` en el momento de producirse el *timeout*:
     `ssthresh := máx(flightSize/2, 2*MSS)`

     siendo `flightSize` los "bytes en vuelo", es decir, los enviados y no asentidos hasta ese momento
  2. Se aplica ***Slow Start*** con `cwnd=1`.
  3. Se va doblando `cwnd` cuando se reciben todos los ACKs de los paquetes enviados según `cwnd` (o se suma 1MSS por cada ACK nuevo recibido) hasta que `cwnd` llegue al valor `ssthresh`.
  4. A partir de ese momento se aplica ***Congestion Avoidance***.

Contenidos

Congestion Avoidance (CA)

- Se entra en este modo cuando `cwnd > ssthresh`, es decir, se está transmitiendo cerca de la capacidad de la red y por lo tanto se aumentará más lentamente el valor de `cwnd`.
- En el modo CA:
  - Se incrementa `cwnd` en un 1 MSS al recibir los ACKs de **todos los paquetes enviados según** `cwnd` (aprox. cada RTT): incremento aditivo (Additive Increase, AI)
  - Dicho de otra forma, la `cwnd` se incrementa en $\frac{1 \text{ MSS}}{\text{cwnd}}$ cada vez que se recibe un ACK



OBSERVA: En CA cada ACK recibido permite enviar 1 nuevo segmento, el que ocupa el sitio que deja el paquete asentido, y cada RTT se podrá enviar otro paquete extra por el aumento de cwnd en 1 MSS.

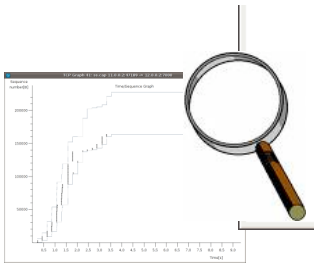
Slow Start y Congestion Avoidance

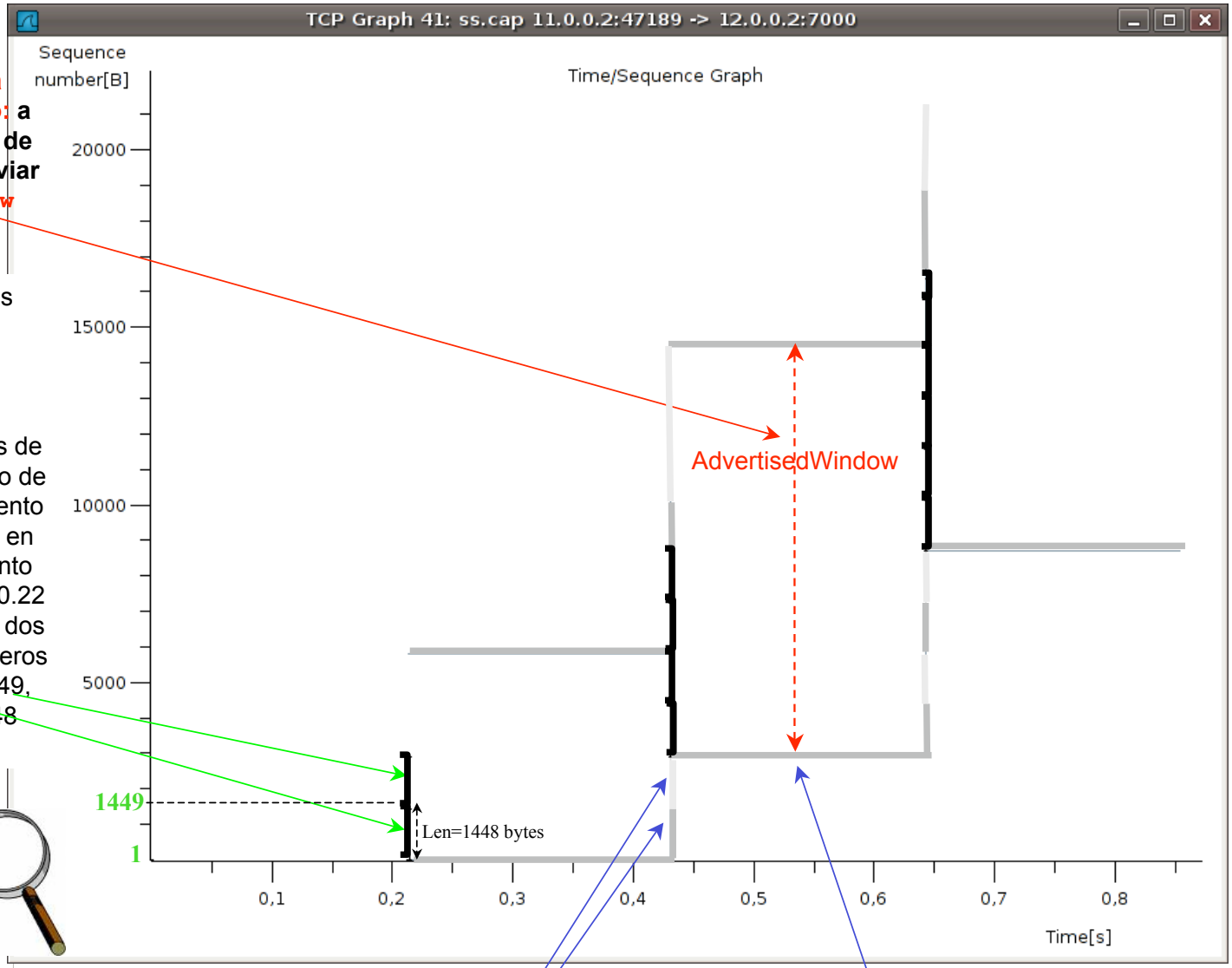
- **SS se utiliza para alcanzar rápidamente la capacidad de la red, cuando se está lejos de ella**
  - En SS el crecimiento de la ventana de control de congestión es multiplicativo.
    - Se denomina *slow* porque, si no hubiera control de congestión, el emisor podría enviar inicialmente la cantidad de datos indicada por la Ventana anunciada (o de flujo). SS es más lento que TCP sin control de congestión, y de ahí el nombre.
  - Cuando hay *timeout*, se aplica SS con `cwnd=1 MSS`, lo que nos permite alejarnos de golpe de situaciones de congestión y crecer rápidamente hasta la mitad de su antiguo valor.
    - Es importante que los *timeout* no venzan antes de tiempo ya que se baja drásticamente el valor de `cwnd` a 1 MSS.
- **CA se utiliza cuando se transmite cerca de la capacidad de la red**
  - En CA el crecimiento de la ventana de control de congestión es aditivo.
  - CA es más lento que SS. No se utiliza CA al iniciar una conexión o al producirse un *timeout* porque con este mecanismo se tardaría mucho más tiempo en descubrir la capacidad de la red.

Gráfica de tcptrace dentro de Wireshark

Ventana anunciada por el otro extremo: a partir del último nº de ACK se pueden enviar `AdvertisedWindow` bytes

Segmentos verticales negros: **segmentos TCP de datos enviados**. Cada segmento ocupa un conjunto de números de secuencia: el número de secuencia del segmento TCP más la longitud en bytes de ese segmento TCP. En el instante 0.22 segundos se envían dos segmentos con números de secuencia 1 y 1449, cuya longitud es 1448 bytes.

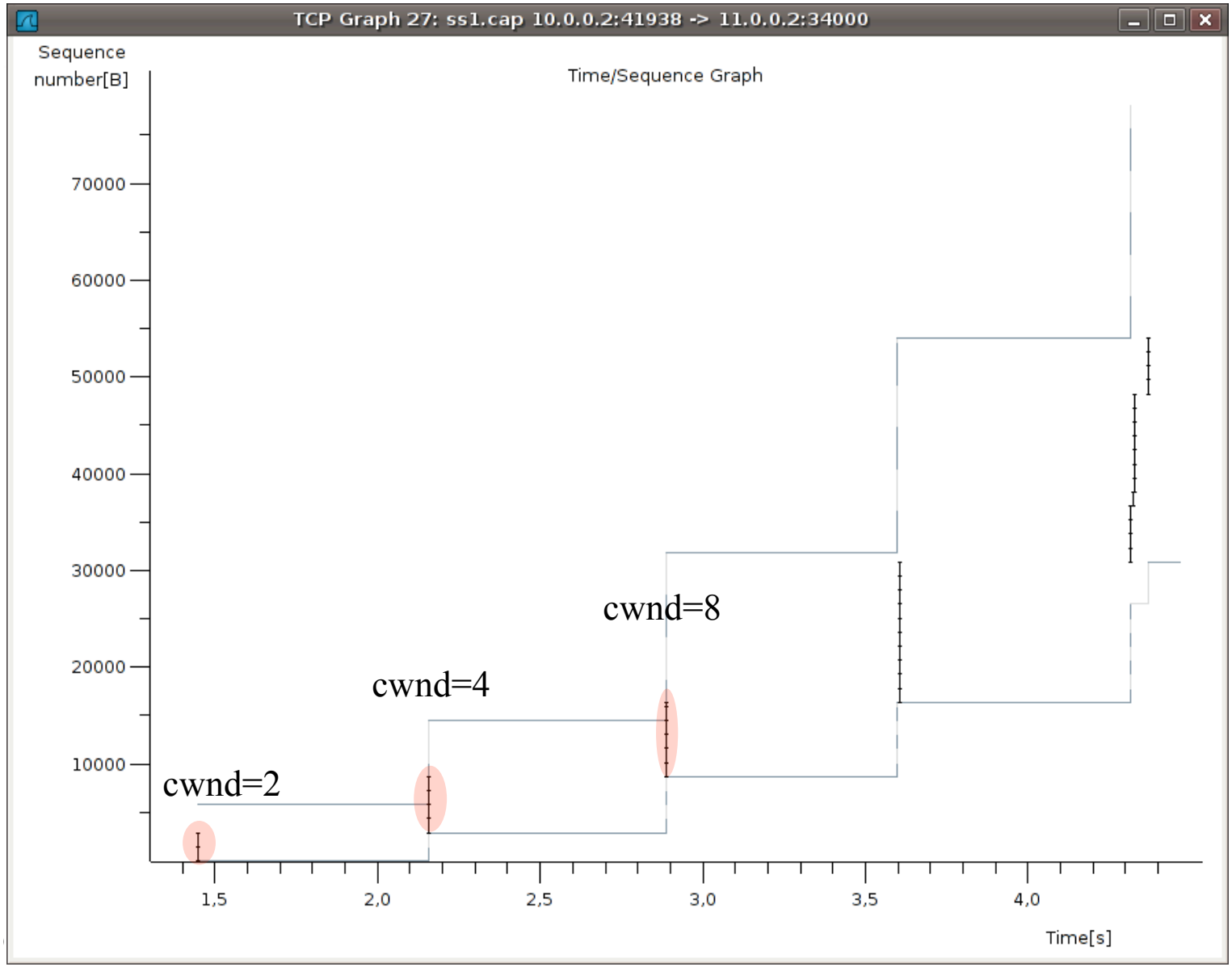




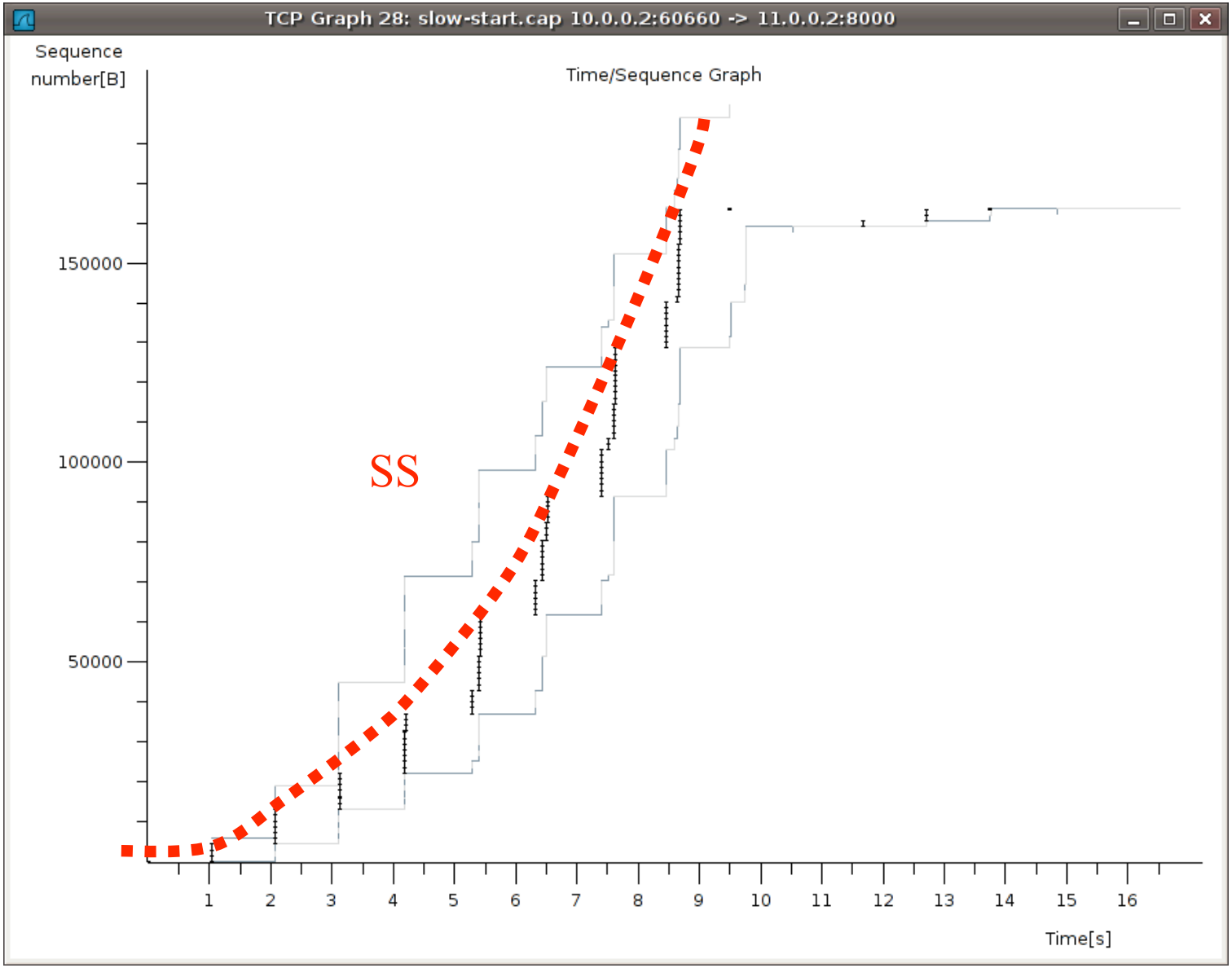
Segmentos verticales grises: **segmentos TCP de ACK** recibidos

Línea inferior gris claro: **último nº de ACK recibido**

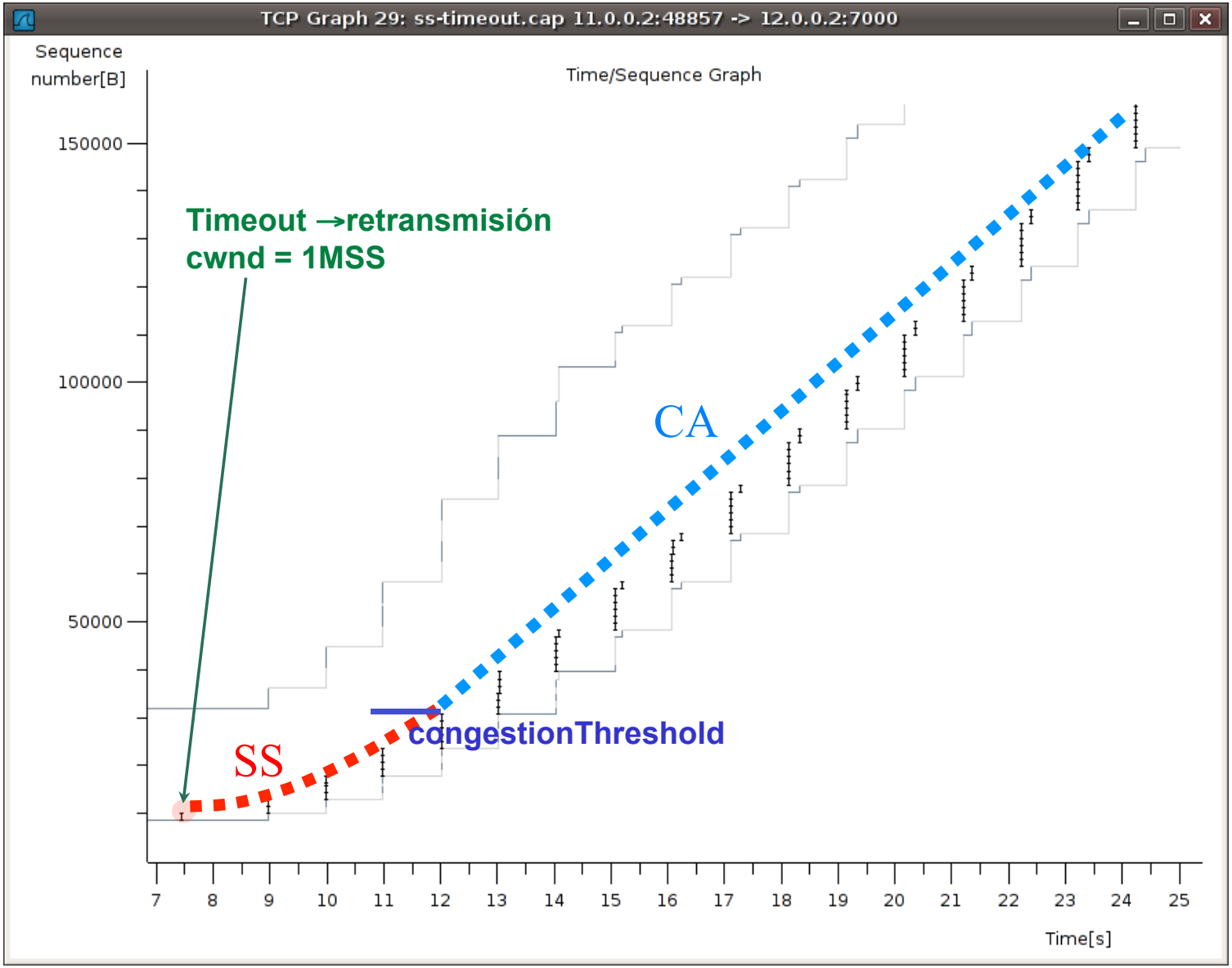
Slow Start


TCP Graph 27: ss1.cap 10.0.0.2:41938 -> 11.0.0.2:34000
Sequence number[B]
Time/Sequence Graph
70000
60000
50000
40000
30000
20000
10000
cwnd=2
cwnd=4
cwnd=8
1,5
2,0
2,5
3,0
3,5
4,0
Time[s]

Slow Start


TCP Graph 28: slow-start.cap 10.0.0.2:60660 -> 11.0.0.2:8000
Sequence number[B]
Time/Sequence Graph
150000
100000
50000
SS
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Time[s]

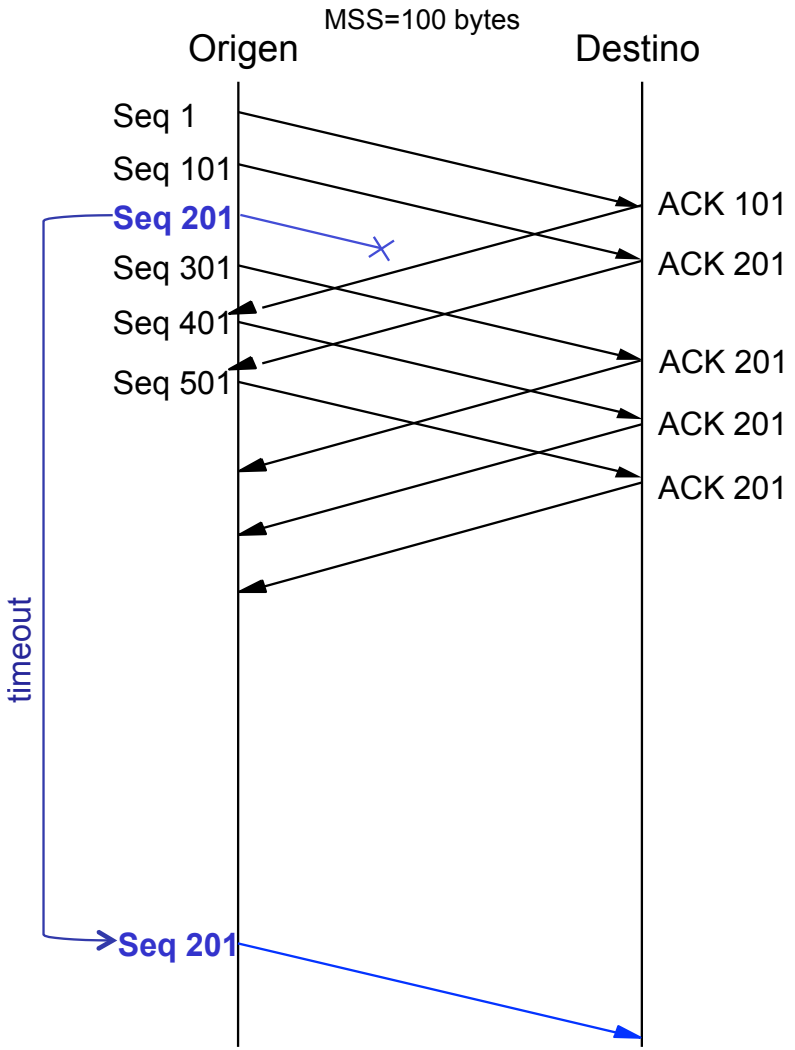
Slow Start y Congestion Avoidance



Contenidos

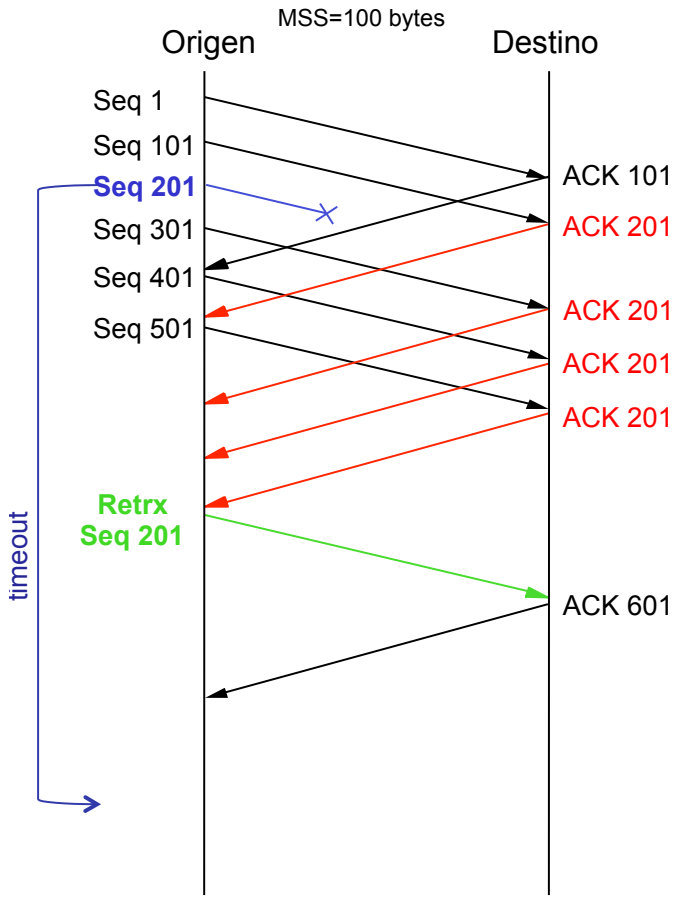
Motivación para Fast Retransmit

- Los *timeouts* provocan períodos grandes de inactividad.
- Tras la pérdida aún no detectada de un paquete (después de enviar el segmento número 201) se siguen enviando tantos paquetes como permite la ventana efectiva, hasta que se hace cero (después de enviar el segmento número 501). Desde ese momento hasta que vence el *timeout* el emisor se para: no se puede enviar nada nuevo.



Fast Recovery

- ***Fast Retransmit***: se utiliza la llegada de 3 ACKs repetidos (3 repetidos, en total 4 ACKs del mismo paquete) como señal de que ha habido una pérdida (y por lo tanto congestión) y se retransmite sin esperar a que venza el *timeout*.
  - Menos de 3 ACKs repetidos se considera que pueden deberse a desorden en los segmentos enviados, no a una pérdida
- Mejoras que aporta *fast retransmit*:
  - Se retransmite antes
  - Se evita que se produzca el *timeout*, y por tanto no se baja a cwnd=1.
  - Se aprovecha más la capacidad de la red: No se vacía la red, sino que se siguen enviando mensajes y recibiendo ACKs



Con *Fast Retransmit* no se espera al *timeout* para retransmitir

Contenidos

Fast Recovery

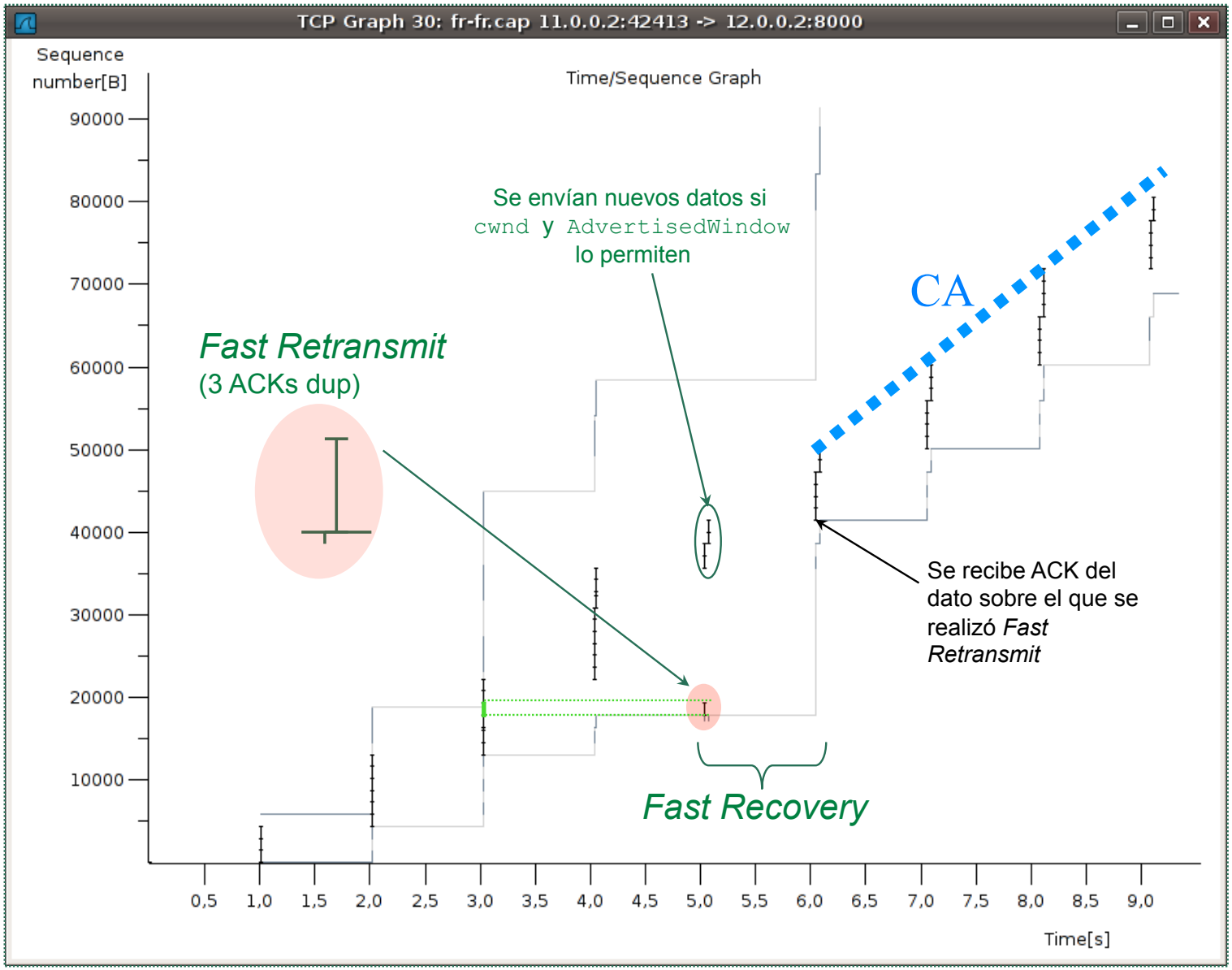
- Después de realizar un *Fast Retransmit* se aplica un nuevo modo de control de congestión llamado ***Fast Recovery***:
  - Se calcula `ssthresh` como la mitad del `flightSize` que había cuando se realizó la retransmisión rápida:
    `ssthresh := máx(flightSize/2, 2*MSS)`
  - **Hasta que no se reciba el ACK del paquete retransmitido**, por cada ACK repetido que llegue se aumenta `cwnd` para que se puedan enviar datos nuevos (si `AdvertisedWindow` lo permite):
    - Cada ACK duplicado recibido significa que un segmento nuevo ha llegado al receptor. Por tanto, un segmento de datos ha "abandonado la red" y la red puede admitir un nuevo segmento.
    - En Fast Recovery se comienza con `cwnd` que tiene en cuenta los 3 ACKs duplicados
      `cwnd := ssthresh + 3*MSS`
    - Por cada ACK duplicado adicional `cwnd := cwnd + MSS`

    NOTA: Si esto no se hiciera, no se podría enviar el segmento "habitual" que corresponde a cada ACK recibido, pues estos ACKs duplicados no liberan realmente un hueco en la ventana, a diferencia de los ACKs "normales".
  - **Cuando llegue el ACK del paquete retransmitido**, se pasa a CA comenzando con `cwnd=ssthresh`.
    - Si el ACK no llega, es que hay un problema de congestión grave. Vencerá el *timeout* y se pasará a *Slow Start* con `cwnd`=1MSS hasta llegar a `máx(FlightSize/2, 2*MSS)` (con `FlightSize` calculado en el momento en el que se produjo el *timeout*), y a partir de ahí se pasará a CA.

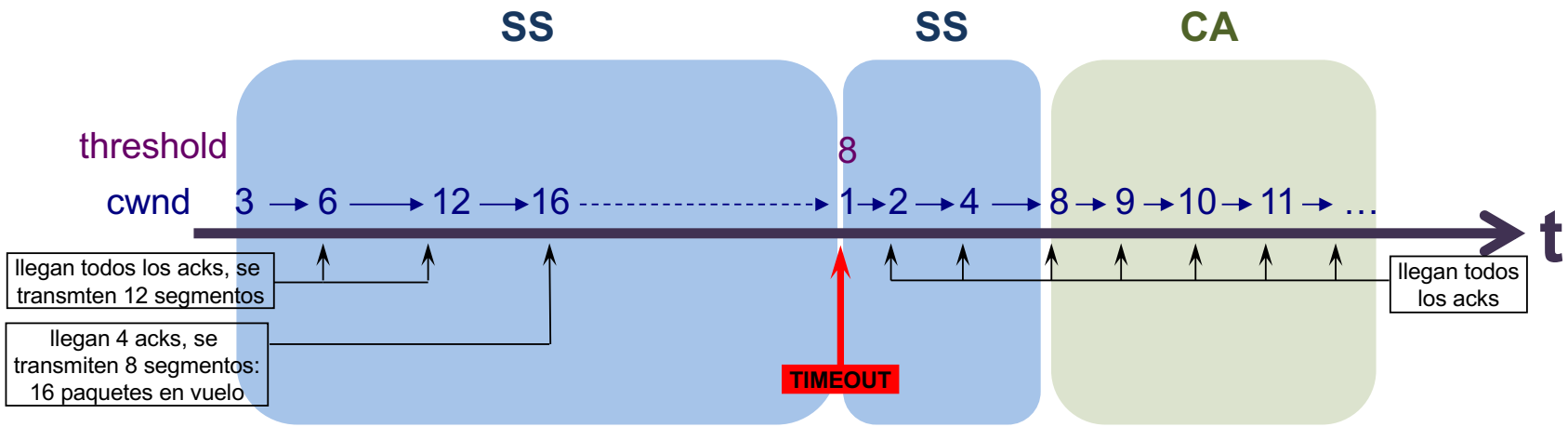
Pseudocódigo de Fast Retransmit y Fast Recovery

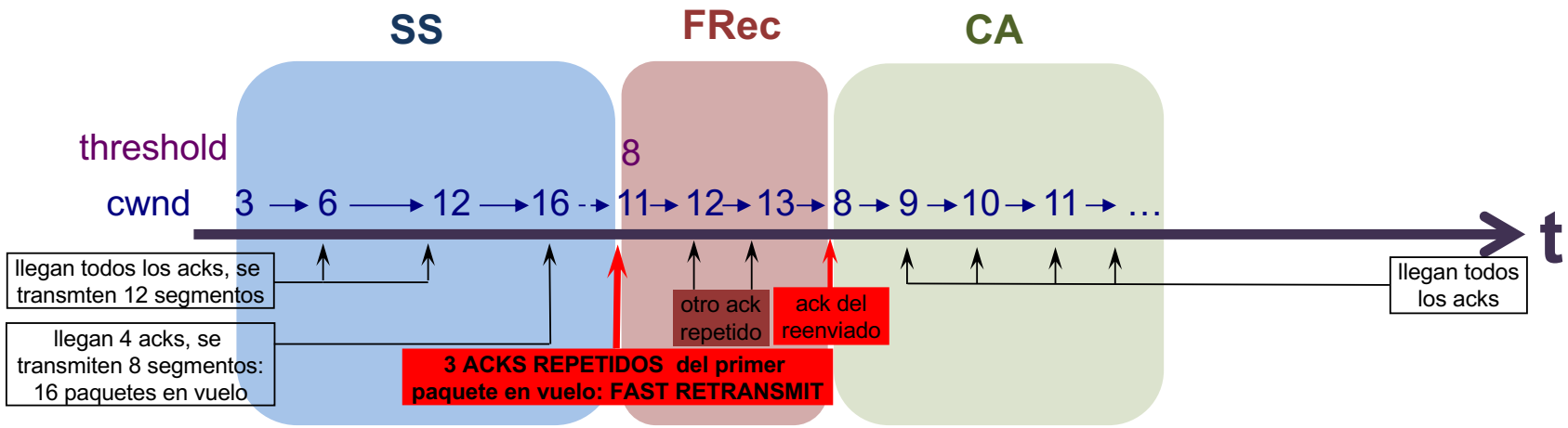
```
if Number of DUP-ACK received = 3
        ssthresh :=  FlightSize / 2;
        Retransmisión del segmento perdido;
        CWND := ssthresh + 3 ; -- 3 por los 3 DUP-ACKs

for each next DUP-ACK
        -- Aumenta CWND para reflejar que
        -- están llegando segmentos al receptor
        CWND := CWND + 1;

if permitido por CWND y Advertised Window
        Transmitir un segmento nuevo;     -- si hubiera alguno

if Not DUP-ACK                            -- i.e. ACK del paquete retransmitido
        CWND := ssthresh;       -- baja la CWND
        Paso a Congestion Avoidance;
```

Fast Retransmit y Fast Recovery



Comparación del Control de Congestión tras *Timeout* y tras *Fast Retransmit*





Contenidos

Contenidos

Tahoe, Reno

- TCP Tahoe (BSD Network Release 1.0, BNR1):
  - *Slow Start*
  - *Congestion Avoidance*
- TCP Reno (BSD Network Release 2.0, BNR2):
  - *Slow Start*
  - *Congestion Avoidance*
  - *Fast Retransmit y Fast Recovery*.

  TCP Reno no se comporta bien cuando hay múltiples pérdidas de paquetes.

Contenidos

TCP New Reno

- *Slow Start*
- *Congestion Avoidance*
- *Fast Retransmit y Fast Recovery*, no se abandona Fast Recovery hasta que todos los paquetes enviados en el momento en el que se realizó la retransmisión rápida estén asentidos. Con la recepción de un ACK parcial:
  - Se retransmite el segmento que faltaba y `cwnd=cwnd-nACK+1`, donde `nACK` es el número de segmentos nuevos asentidos (para que cuando FRetx termine la cantidad de segmentos en camino sea parecida a `ssthresh`).
  - Si `cwnd` lo permite, se envían más segmentos.
- La implementación de TCP por defecto en NetGUI es New Reno.

Contenidos

Extensión de New Reno con SACKs

- **Problema**: La información aportada por los ACKs acumulativos es muy limitada.
- **Alternativa**: *Selective Acknoledgement* (SACK) se utiliza para asentir datos no contiguos, de forma que el emisor pueda saber qué datos son los que realmente no ha conseguido el receptor.
- Para poder utilizar SACK, emisor y receptor deben estar de acuerdo y lo notificarán en el campo opciones del establecimiento de la conexión TCP.
- No cambia el significado del campo número de asentimiento que viaja en la cabecera TCP.
- Si se decide usar SACK, la información del asentimiento selectivo se encontrará disponible en las opciones de los segmentos TCP que se envíen una vez iniciada la conexión. Esta información incluye:
  - Nombre de la opción: TCP SACK Option (5)
  - Longitud de la opción
  - Lista de bloques asentidos. Cada bloque se especifica con la pareja de números de `[nSeq1, nSeq2]`, lo que significa que se han recibido todos los datos correspondientes a los números de secuencia desde `nSeq1` hasta `nSeq2-1`.

Algoritmo TCP New Reno con SACKs

- Durante Fast Recovery se mantiene una nueva variable `pipe` que representa una estimación de los paquetes que se encuentran en camino (no han llegado al receptor). Se tiene en cuenta el número de ACK y los bloques que se asienten en SACK.
- Por tanto, con el tercer ACK duplicado:
  - `pipe=flightSize-3MSS`, debido a que 3 segmentos han abandonado la red
  - Se realiza la retransmisión rápida del primer segmento no asentido
  - `ssthresh=flightSize/2`
  - `cwnd=ssthresh`
- Si se recibe otro ACK duplicado, significa que otro segmento de los enviados ha abandonado la red:
  - `pipe-= 1MSS`
  - Si `pipe<cwnd` envío paquetes nuevo/retransmisión (por cada paquete `pipe+=1MSS`). TCP mantiene información de que paquetes se han asentido a través de SACK.
- Si se recibe un ACK parcial (no asiente todo lo que teníamos pendiente cuando empezó Fast Recovery):
  - `pipe = pipe - 2MSS`
    - El valor 2MSS: uno por el paquete que se ha retransmitido y que ahora ha llegado el asentimiento y otro por el paquete original (que se supone que se ha perdido).
  - Si `pipe<cwnd` envío paquetes nuevo/retransmisión (por cada paquete `pipe+=1MSS`). TCP mantiene información de que paquetes se han asentido a través de SACK.

Contenidos

Alternativas para evitar la congestión: TCP Vegas

- Reno/New Reno fuerzan la situación de congestión, aumentando `cwnd` hasta que hay pérdidas. En el momento en el que se produce una pérdida reduce `cwnd` ya sea porque se activa SS (ha habido timeout) o se activa FReco/FRetx (ha habido 3 ACKs duplicados).

- **TCP Vegas** intenta medir la capacidad esperada de la red (`Expected`) y comparar con la capacidad que se está obteniendo (`Actual`).

  `Expected = cwnd/BaseRTT`, siendo `BaseRTT` el mínimo de los RTTs medidos para una conexión.

  `Actual = cwnd/RTT`, siendo `RTT` el RTT medido para los datos enviados y sus ACKs recibidos.

  - Si la capacidad actual está lejos de la esperada, es porque se están midiendo `RTT >> BaseRTT`, la red está cercana a la congestión, y por ello, se disminuye la ventana.
  - Si la capacidad actual está cerca de la esperada, es porque `BaseRTT` y `RTT` son parecidos, se aumenta la ventana porque la red parece no estar congestionada.

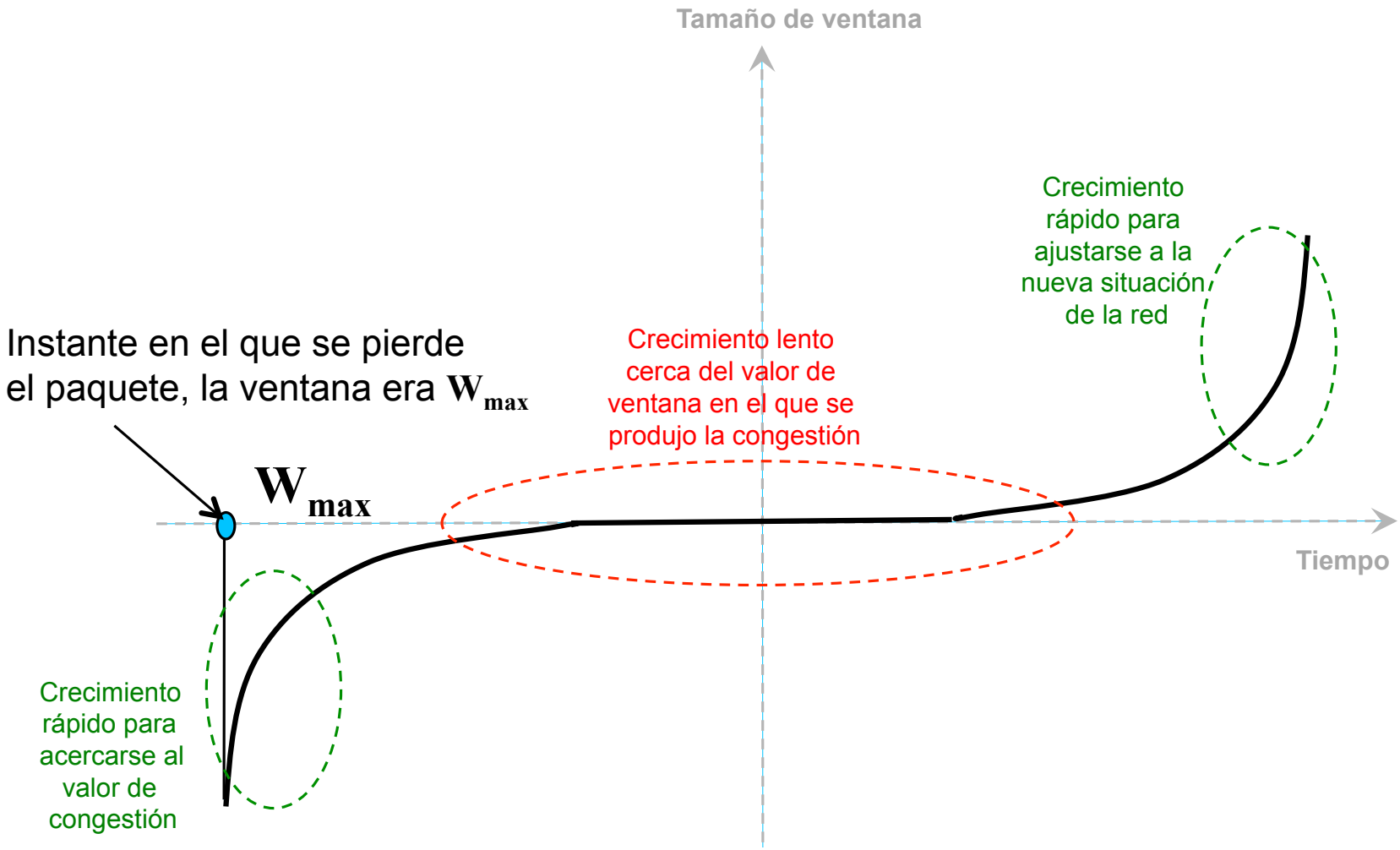
Contenidos

Alternativas para redes de ↑ ancho de banda y ↑ retardo: TCP Cubic

- El incremento de la ventana no depende de RTT (que es alto).
  - Si ↓ RTT, TCP New Reno funciona bien ya que los ACKs se reciben rápidamente y cwnd aumenta rápidamente.
- En TCP Cubic el incremento de ventana depende del tiempo transcurrido desde la última pérdida.
- Cuando hay retransmisión, se almacena el valor de la ventana en ese momento W_{max} y se calcula un nuevo valor:
  $W = W_{max}(1 - \beta)$, valor típico $\beta = 0{,}2$ (reducción de 80 %)
- La ventana aumenta según la siguiente fórmula:
  $W(t) = C(t - K)^3 + W_{max}$, donde $K = \sqrt[3]{\frac{W_{max}\beta}{C}}$ y C parámetro de Cubic

  O lo que es lo mismo:
  - Aumento rápido de la ventana para recuperar pronto el ancho de banda
  - Cuando la ventana se acerca a W_{max} se ralentiza el incremento de ventana para ver si no hay pérdidas utilizando el valor de ventana que existía cuando se produjo el último timeout.
  - Si no hay pérdidas, se inicia el crecimiento rápido de la ventana. Parece que la red ya no se encuentra congestionada y hay que calcular el tamaño óptimo.

Crecimiento de la ventana en TCP Cubic



Implementación que actualmente usa Linux.

Contenidos

Referencias

- RFC 5681, **TCP Congestion Control**, 2009: `http://www.faqs.org/rfcs/rfc5681.html`
- RFC 6582, **The NewReno Modification to TCP's Fast Recovery Algorithm**, 2012: `http://www.faqs.org/rfcs/rfc6582.html`
- RFC 2018, **TCP Selective Acknowledgment Options**, 1996: `http://www.faqs.org/rfcs/rfc2018.html`
- RFC 6675, **A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP**, 2012: `http://www.faqs.org/rfcs/rfc6675.html`