# Control 2
# CC4303-2, Redes

José Espina
joseguillermoespina@gmail.com

## 1. Capa de transporte

### 1.1. Pregunta

Estudiar cómo se comportan los algoritmos *Stop & Wait*, *Go-Back-N* y *Selective Repeat* en un entorno de pérdida 0.2 y delay alto (0.5 segundos de delay, 1 segundo de *RTT*). Para estudiar cada algoritmo puede usar el simulador del curso o puede hacerlo por su propia cuenta. A partir de lo observado en su estudio proponga un algoritmo y tamaño de ventana óptimos para los parámetros dados. Explique cómo estudió los algoritmos y justifique su respuesta

### 1.2. Respuesta

#### 1.2.1. Contexto

Se definen los algoritmos de la pregunta, más un cuarto que aparece en el simulador del curso, los cuales difieren en términos de eficiencia, complejidad, y requerimientos de *buffer* (basado en el capítulo 3.4 de [1])

**Stop & Wait** El transmisor envía un *frame* y espera el *acknowledgments* del receptor, antes de enviar el siguente. Este algoritmo es muy lento y no saca provecho del ancho de banda disponible

**Go-back-n** Los *frames* subsecuentes a uno dañado se descartan, sin enviar *acknowledgments* de éstos. Éste acercamiento usa todo el ancho de banda disponible, pero los desperdiciará en un alto porcentaje si la tasa de error es alta

**Selective repeat** Permite al receptor aceptar y almacenar los *frames* que le siguen a uno dañado o perdido. Transmisor y receptor acuerdan mantener una ventana de transmisión de ancho fijo, que permitirá enumerar los *frames* por enviar y recibidos respectivamente. La implementación de la ventana requerirá de un *buffer* por parte de ambos

**Selective repeat + CACK (del inglés, Cumulative Acknowledgments)** Es el mismo *Selective repeat* con la ventaja de que el receptor puede enviar un sólo *frame* de tipo *acknowledgment* para avisarle el emisor de todos los *frames* recibidos satisfactoriamente dentro de la ventana de transmisión

#### 1.2.2. Experimento

Se experimentó con los 4 algoritmos utilizando la versión 4 del simulador editado por "Jo" Piquer[2] con los siguientes parámetros generales. Se ejecutó cada algoritmo hasta un envío aproximado de 700 paquetes

En la tabla a continuación se encuentran los parámetros generales para los 4 algoritmos

Los parámetros *end-to-end delay* y *loss probability* son inidicados en el enunciado del problema. *Timeout* es de 500 milisegundo sobre el *RTT*, lo que debiese darle tiempo suficiente para recibir el *ACK*, o de descartarlo en caso de no llegar hasta ese momento. Se escogió el máximo valor en el envío de

Cuadro 1: Parámetros generales para experimentar con los 4 algoritmos

| Nombre del parámetros | Valor |
|---|---|
| *Windows size* | 15 (salvo *Stop & Wait*: 1) |
| *End-to-end delay* | 500 |
| *End-to-end delay variance* | 0 |
| *Timeout* | 1500 |
| *Number of packets emited per minute* | 120 (valor máximo) |
| *Loss probability* | 0.2 |

paquetes por minuto, esperando poner a prueba el desempeño de cada algoritmo. El parámetro *end-to-end delay variance* está fuera del alcance en este ejercicio. Finalmente, el tamaño de la ventana *windows size*, fue configurado en 15, de manera arbitraria, salvo para implementar *Stop & Wait*, donde se debe fijar en 1.

En las siguientes tablas, se presentan los resultados para *Stop & Wait*, *Go-Back-N*, *Selective Repeat*, y *Selective Repeat + CACK*. Los valores se aproximaron a 2 decimales

Cuadro 2: Resultado del experimento para los 4 algoritmos

| | *Stop & Wait* | *Go-Back-N* | *Selective-Repeat* | *Selective Repeat + CACK* |
|---|---|---|---|---|
| Total packets sent | 698 | 700 | 697 | 697 |
| Total OK | 315 | 301 | 320 | 338 |
| Useful BW (packets/s) | 0.21 | 0.25 | 0.29 | 0.30 |
| Total BW (packets/s) | 0.46 | 0.59 | 0.64 | 0.62 |
| Current BW (packets/s) | 0.63 | 1.23 | 1.24 | 0.68 |
| Loss Prob | 0.2 | 0.22 | 0.28 | 0.2 |

A continuación, se presentan los gráficos de línea resultantes de cada simulación. NOTA: Las escalas de los ejes no son iguales para los 4 gráficos, por lo que no son comparables

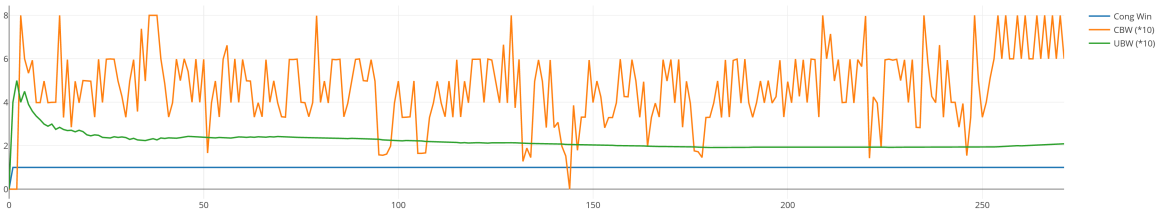



Figura 1: *Stop & Wait*

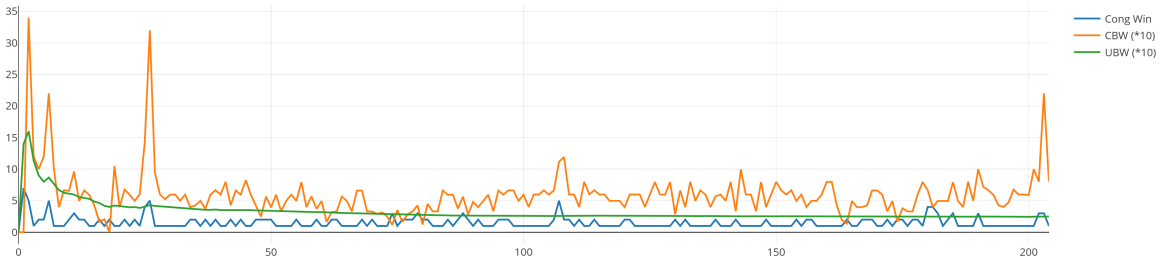



Figura 2: *Go-Back-N*

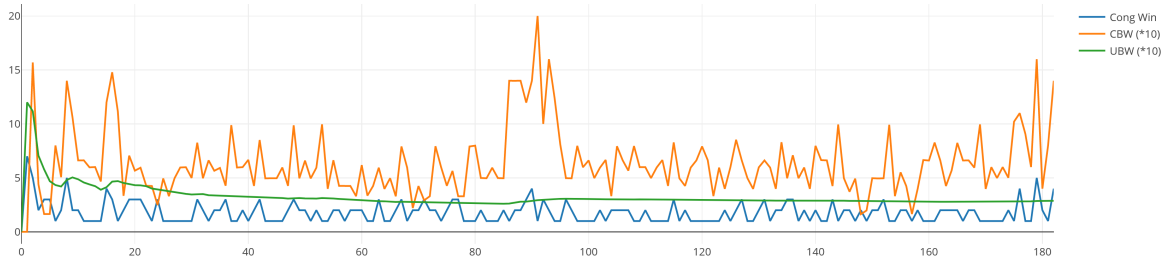



Figura 3: *Selective-Repeat*

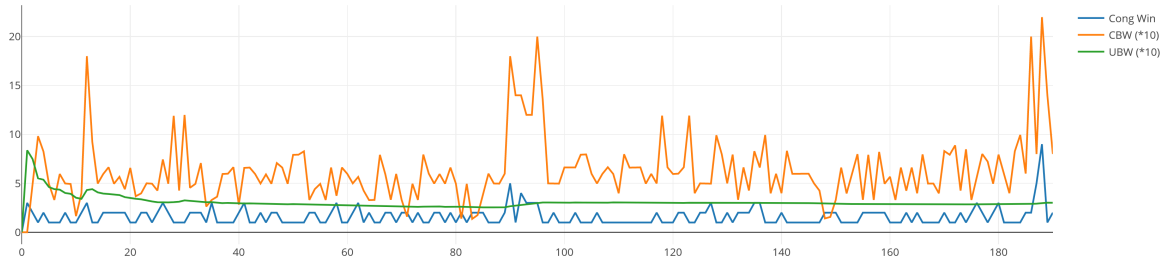



Figura 4: *Selective Repeat + CACK*

### 1.2.3. Análisis de los resultados

Se puede apreciar, que *Selective-Repeat*, y su par con *CACK*, tuvieron los mejores desempeños con 338 y 320 *frames* enviados exitosos respectivamente, y un ancho de banda útil de un 30 % para ambos aproximadamente. Si bien *Selective-Repeat* con *CACK* logra el mismo rendimiento que *Selective-Repeat*, éste consume menor cantidad de ancho de banda ya que necesita sólo 1 *frame ACK* para confirmarle al emisor que todos fueron recibidos exitosamente posterior a un fallo, lo que lo hace más eficiente

Ningún gráfico lo refleja, pero durante el experimento se pudo observar que el gran tamaño de ventana de transmisión con el que se configuró el simulador se desperdició, debido a que nunca se usaron más allá de 4 o 5 *frames* del *buffer*. Ésto se debe a que el *RTT* es bajo dentro dentro del simulador (es, de hecho, el valor más pequeño que es posible configurar). Por la misma razón, se puede ver en los gráficos de línea, que la ventana de congestión se mantiene relativamente baja y estable. Se hubiese sacado más provecho al tamaño de ventana si el *RTT* hubiese sido mayor. Por otro lado, la ventana de congestión hubiese tenido mayores valores en un ambiente con un poco más de pérdida y/o mayor *RTT*

### 1.2.4. Conclusión

Para el escenario de la pregunta, el algoritmo con mejor desempeño (mejor ancho de banda útil logrado), y, además, de mayor eficiencia (menor uso de ancho de banda), es *Selective-Repeat* con *CACK*

# 2. Capa de red

## 2.1. Pregunta

El *AS Hijacking* hace referencia a cuando un sistema (*hijacker*), originalmente ajeno a la red, se posiciona de tal forma que puede ver pasar los mensajes que van de un nodo a otro sin ser identificado. Si los mensajes que ve el *hijacker* se envían de forma insegura, este podrá ver su contenido permitiéndole, por ejemplo, robar claves de Internet. Utilizando la configuración de nodos y el código necesario para la actividad de la semana 13-14 simule *AS hijacking*. Para ello debería insertar un nuevo nodo n de tal forma que los mensajes que van desde 8887 a 8881 pasen por n antes de llegar a 8881. Haga que n además imprima todos los mensajes que pasen por él. Note que para hacer que los mensajes de 8887 a 8881 pasen por n, debe insertar a n junto a sus propias tablas de ruta en algún lugar de la red y luego correr el algoritmo de ruteo y esperar a que se estabilice. Explique cómo y porqué su *hijacking* fue exitoso.

## 2.2. Respuesta

### 2.2.1. Contexto

En el RFC 7132 "Modelo de amenaza para enrutamiento seguro de BGP" [3], página 9, se describen los ataques en *routers* en BGP externa. Entre ellos: *AS Insertion*, *False (Route) Origination* (1), *Secure Path Downgrade*, *Invalid AS_PATH Data Insertion*, *Stale Path Announcement*, *Premature Path Announcement Expiration*, *MITM (Man-In-The-Middle) Attack* (2), *Compromised Router Private Key* y *Withdrawal Suppression Attack*.

Lo descrito en la pregunta se podría lograr con los ataques (1) y (2) en conjunto. En (1) un *router* atacante origina una ruta para un prefijo del AS víctima (donde, obviamente, no está autorizado), desviando el tráfico a su propio AS. Esto funcionaría sacando provecho a que los *router* derivan la comunicación primero a rutas donde prefijos declarados son más específicos, y engañando, de manera administrativa, con una solicitud a los AS colindantes. En ese momento, el atacante tiene 2 opciones: podría no reenviar la comunicación creando un *blackhole*, denegando el servicio, tal como lo que sucedió, por accidente, con el conocido caso de cuando el gobierno de Pakistán censuró, al interior de su país, al sitio YouTube, generando una caída del servicio en todo el mundo por 2 horas aproximadamente. La segunda opción es que, posterior a algún análisis malicioso, manipulación de datos o generación de *spam*, el atacante reenvíe la comunicación a la víctima, esperando que no se entere que su tráfico fue desviado y analizado. Esto último sería clasificado como el ataque tipo (2)

Un muy buen ejemplo de esto último se puede apreciar en la presentación *"Stealing The Internet"* realizada en la Defcon '16[4] y que inspiró a hacer el siguiente experimento

### 2.2.2. Experimento

Se implementó un *script* router_normal.py para simular el comportamiento de un router AS. Se implementó en Python 3, con el módulo SOCKETSERVER para conexiones entrantes y SOCKET para redirigir la comunicación en caso que el mensaje no fuese para el *router*

La idea ejecutar un script para cada *router* de la simulación. Al iniciar, se carga la tabla de rutas desde un archivo de texto, en el formato de la actividad de la semana 11-12 de la capa de redes (Red (CIDR), puerto inicial, puerto final, IP para llegar y puerto para llegar). El script 2 parámetros, el puerto en que va a escuchar conexiones entrantes, y la ruta del archivo que contenga la tabla de rutas

Los supuestos dentro de la implentación son:

1. Se debe ejecutar el *script* con la versión 3, o superior, de Python
2. Como IP siempre se usa 127.0.0.1. Sería interesante otros rangos para hacer un experimento más complejo
3. Se usa el programa NetCat para experimentación
4. El mensaje de entrada, mediante NetCat, del script es de sintaxis correcta, cumpliendo siempre con el formato "IP, PUERTO, MENSAJE"
5. Tanto las rutas como los archivos de tablas de rutas existen y están correctos
6. El límite del *buffer* de entrada es de 256 caracteres
7. Se usó protocolo TCP de manera arbitraria

Para el experimento realizado se usaron 3 *routers* normales y un atacante (el *AS hijacker*). Éste último tiene su implementación propia, en el *script* router_atacante.py. Este *script* es más rígido, en el sentido que abre el puerto 10000 de manera arbitraria cuando se le ejecuta y no tiene archivo de configuración

La siguiente tabla y figuras representan las configuración y visualizan, respectivamente, el ataque que se simulará, donde el *router* víctima es **R1**, y el atacante, que espiará su comunicación desde R3 sin que la víctima se entere, es **Atacante**

Cuadro 3: Puertos de los AS de la simulación

| | Router AS 1 | Router AS 2 (R2) | Router AS 3 (R3) | Router AS atacante |
|---|---|---|---|---|
| Nombre | R1 | R2 | R3 | R Atacante |
| Puerto | 9000 | 9001 | 9500 | 10000 |

Cuadro 4: Configuración de R1 (archivo rutas_R1.txt)

| CIDR | Puerto inicio | Puerto Final | IP de llegada | Puerto de llegada |
|---|---|---|---|---|
| 127.0.0.0/24 | 9001 | 9499 | 127.0.0.1 | 9001 |
| 127.0.0.0/24 | 9500 | 9999 | 127.0.0.1 | 9500 |

Cuadro 5: Configuración de R2 (archivo rutas_R2.txt)

| CIDR | Puerto inicio | Puerto Final | IP de llegada | Puerto de llegada |
|---|---|---|---|---|
| 127.0.0.0/24 | 9000 | 9000 | 127.0.0.1 | 9000 |
| 127.0.0.0/24 | 9500 | 9999 | 127.0.0.1 | 9500 |

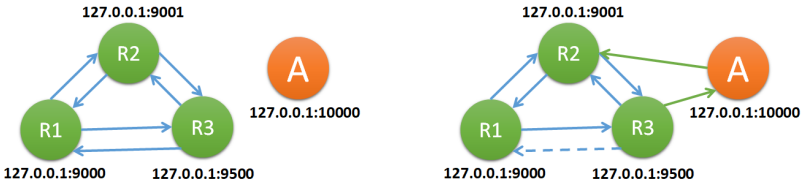
Cuadro 6: Configuración de R3 **antes del ataque** (archivo rutas_R3.txt)

| CIDR | Puerto inicio | Puerto Final | IP de llegada | Puerto de llegada |
|---|---|---|---|---|
| 127.0.0.0/24 | 9000 | 9000 | 127.0.0.1 | 9000 |
| 127.0.0.0/24 | 9001 | 9499 | 127.0.0.1 | 9001 |

Cuadro 7: Configuración de R3 **después del ataque** (archivo rutas_R3_corrupto.txt)

| CIDR | Puerto inicio | Puerto Final | IP de llegada | Puerto de llegada |
|---|---|---|---|---|
| 127.0.0.0/24 | 9000 | 9000 | 127.0.0.1 | 9000 |
| 127.0.0.0/24 | 9001 | 9499 | 127.0.0.1 | 9001 |
| ⋆127.0.0.0/25 | 9000 | 9000 | 127.0.0.1 | 10000 |

En la última tabla, la última configuración ($\star$) es la que permite el ataque, ya que tiene un prefijo más específico que la configuración original que permitía la comunicación directa de R3 a R1 (el de la primera fila). A continuación, una figura que representa la topología de nuestra "internet" simulada antes y después del ataque




(a) Antes del ataque (b) Posterior del ataque

Figura 5: Representación, en grafos, de la topología de la red, antes y después del ataque

A continuación, se describe el algoritmo de los *routers* normales

**Algorithm 1:** Funcionaimento de un router normal de la simulación

```
Input: Comunicación entrante: se reciben destinatario(ip y puerto), y mensaje
Output: Mensaje recibido o retransmitido
Carga de tabla de rutas desde archivo dado por argumento;
Inicialización de matriz de destinos vacía;
Se inicia el servidor de conexiones entrantes;
while escucha nuevas conexiones do
    if ¿El mensaje es para mí? then
        Se lee e imprime;
    else
        if Destinatario ya existe en matriz de rutas then
            Se reenvía usando Round-Robin;
        else
            Se crea la matriz de rutas;
            Se reenvía usando la primera ruta;
        end
    end
end
```

La matriz de rutas se va armando a medida que van llegando los mensajes. Tiene como fin implementar Round-Robin en caso de haber múltples caminos para el mismo destino

Para replicar el experimento, ejecute los siguientes *scripts* en consolas separadas:

```
$ python router_normal.py 9000 rutas_R1.txt
$ python router_normal.py 9001 rutas_R2.txt
$ python router_normal.py 9500 rutas_R3.txt
```

Listing 1: Inicio de routers

Pruebe enviar un mensaje desde cualquier *router* a otro, usando NetCat (nc). Por ejemplo:

```
nc 127.0.0.1 9500 << EOF
127.0.0.1,9000,chupete
EOF
```

Listing 2: Ejemplo de transmisión de mensaje a R1 a través de R3

Para simular el ataque, detenga R3 e inícielo con el archivo de configuración rutas_R3_corrupto.txt

```
python router_normal.py 9500 rutas_R3_corrupto.txt
```

Listing 3: Ejemplo de transmisión de mensaje a R1 a través de R3

Levante el R atacante

```
python router_atacante.py
```

Listing 4: Se levanta el router atacante de R1

Por último, probar que el ataque funciona, envíe, nuevamente, un mensaje a R1 a través de R3. Se debiese ver el mensaje secuestrado, tanto en la consola del R Atacante, como en R1

### 2.2.3. Conclusión

En la realidad, esto se pudo haber logrado engañando a los administradores del AS, solicitando una nueva regla de flujo de datos con un prefijo de CIDR más específico que alguna regla actual. Esto se puede lograr con ingeniería social, registrando un dominio expirado, o suplantando la identidad de un correo (esto último es muy poco probable que funcione hoy en día). Un caso real, fue el denominado *"Incidente LinkTel"*, donde un atacante secuestró a la compañía rusa Link Telecom (AS31733) a través del registro de un dominio DNS recién expirado, y posteriormente, anunciando una serie de prefijos a la compañía Internap (AS12812)[5]

Cabe destacar que éste no es la única forma de ataque en BGP externa, tal como se mencionó al principio, hay varios tipos de ataques y, para esta ocasión, se simuló uno basado en la presentación de la Defcon '16, ya citada anteriormente

# 3. Anexo

Se anexan, a continuación, las simulaciones de la pregunta 1

# Referencias


[1] Tanenbaum, A. S. (2003). Computer networks, fourth edition: Problem solutions. Upper Saddle River, NJ: Prentice Hall PTR.

[2] Simulador de *frames* Johannes Kessler, modificado por José "Jo" Piquer. Retrieved July 10, 2020, from https://users.dcc.uchile.cl/ jpiquer/srgbn4.html

[3] Threat Model for BGP Path Security. (n.d.). Retrieved July 17, 2020, from https://tools.ietf.org/html/rfc7132

[4] Stealing The Internet, An Internet-Scale Man In The Middle Attack Defcon 16. Retrieved July 17, 2020, from https://www.defcon.org/images/defcon-16/dc16-presentations/defcon-16-pilosov-kapela.pdf

[5] Schlamp, J., Carle, G., & Biersack, E. W. (2013). A forensic case study on as hijacking: The attacker's perspective. ACM SIGCOMM Computer Communication Review, 43(2), 5-12.

# Selective Repeat / Go Back N

## configuration

**protocol**

(•) Go back N ( ) Selective Repeat ( ) Selective Repeat+CACK
choosing a new protocol restarts the simulation

**window size**

1
sets the window size for the windows

**end to end delay**

500
time a packet takes from one station to the other

**end to end delay variance (+/- err)**

0
+/- error added to end to end delay

**timeout**

1500

**scroll mode**

Typewriter style
change the style the window scrolls

**number of packets emited per minute**

120
the number of packets the upper layer tries to send per minute

**automatic emission of packets**

start
starts or stops the automatic emission of packets by the upper layer

**loss probability**

0.2
the loss probability of every packet
Total Packets sent:

698

Total OK:

315

Useful BW (packets/s):

0.209431741873716

Total BW (packets/s):

0.46407414548525006

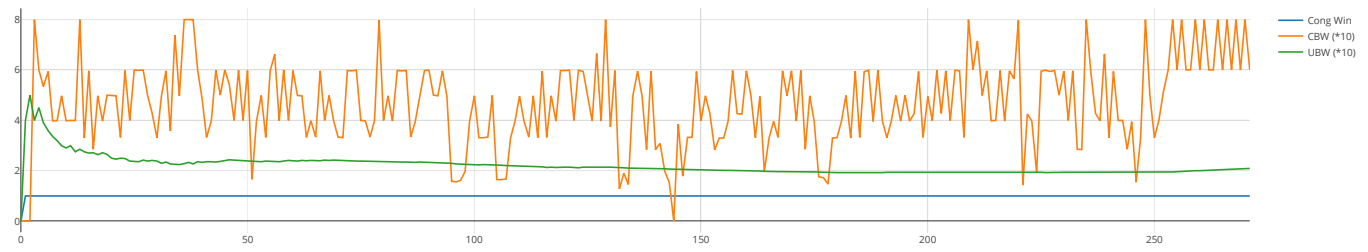
Current BW (packets/s):

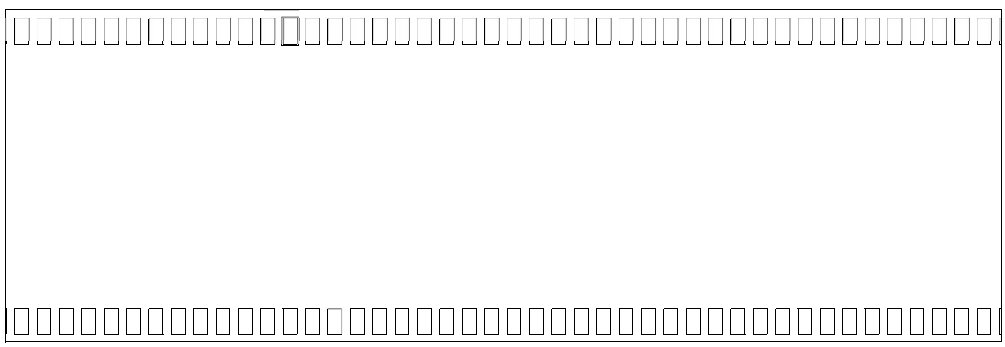
0.6337805006865955

Loss Prob:

0.2

Real-Time Chart with Plotly.js



**legend**

- no data received yet
- data buffered (ready to send, delivered or sent but no ack received yet)
- ack
- cumulative ack
- transmission confirmed
- data has been delivered to upper network layer

coded by Johannes Kessler 2012

small extensions by Jo Piquer 2019

# Selective Repeat / Go Back N

## configuration

### protocol

(•) Go back N ( ) Selective Repeat ( ) Selective Repeat+CACK
choosing a new protocol restarts the simulation

### window size

15
sets the window size for the windows

### end to end delay

500
time a packet takes from one station to the other

### end to end delay variance (+/- err)

0
+/- error added to end to end delay

### timeout

1500

### scroll mode

Typewriter style
change the style the window scrolls

### number of packets emited per minute

120
the number of packets the upper layer tries to send per minute

### automatic emission of packets

start
starts or stops the automatic emission of packets by the upper layer

### loss probability

0.2
the loss probability of every packet
Total Packets sent:

700

Total OK:

301

Useful BW (packets/s):

0.25334162655424247

Total BW (packets/s):

0.5891665733819592

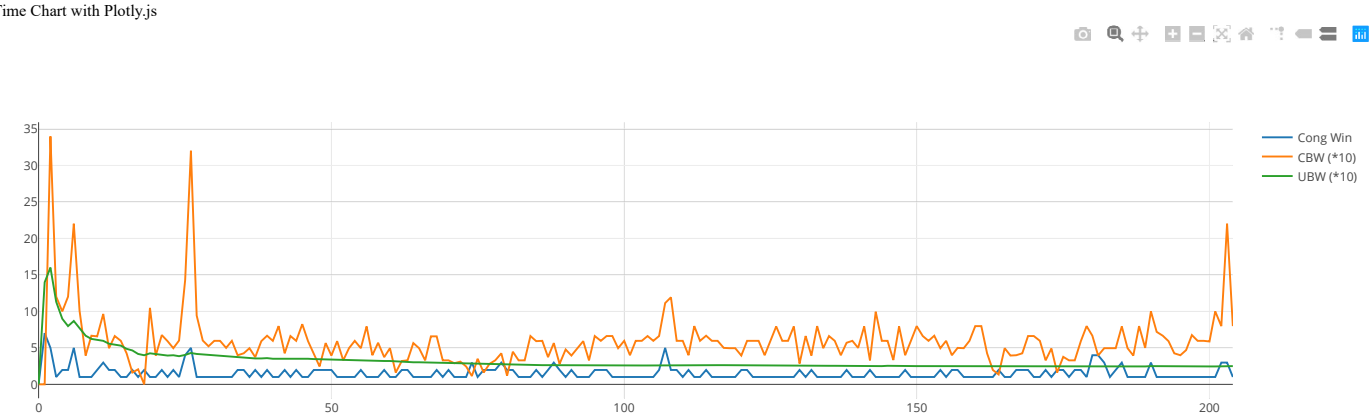
Current BW (packets/s):

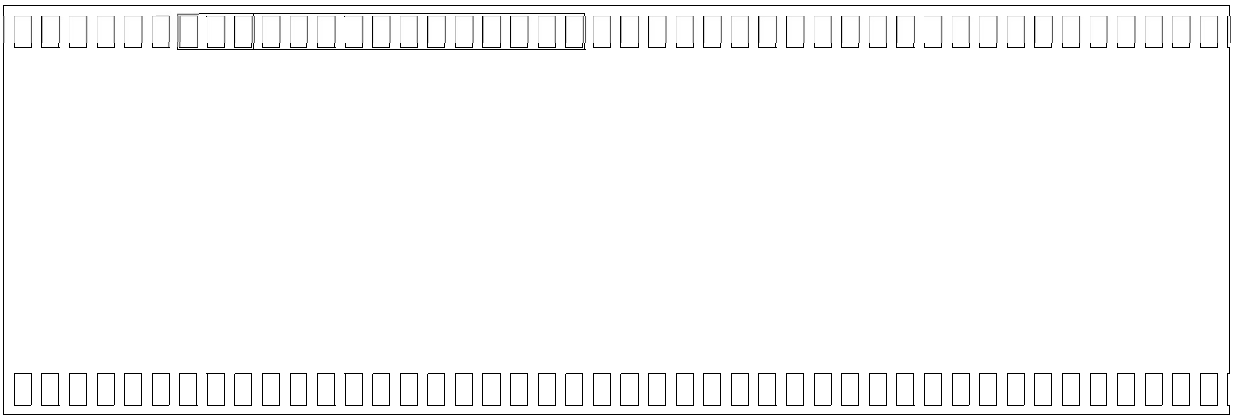
1.2335136160926083

Loss Prob:

0.2237696752341104

Real-Time Chart with Plotly.js





## legend

- no data received yet
- data buffered (ready to send, delivered or sent but no ack received yet)
- ack
- cumulative ack
- transmission confirmed
- data has been delivered to upper network layer

coded by Johannes Kessler 2012

small extensions by Jo Piquer 2019

# Selective Repeat / Go Back N

## configuration

### protocol

( ) Go back N (•) Selective Repeat ( ) Selective Repeat+CACK
choosing a new protocol restarts the simulation

### window size

15
sets the window size for the windows

### end to end delay

500
time a packet takes from one station to the other

### end to end delay variance (+/- err)

0
+/- error added to end to end delay

### timeout

1500

### scroll mode

Typewriter style
change the style the window scrolls

### number of packets emited per minute

120
the number of packets the upper layer tries to send per minute

### automatic emission of packets

start
starts or stops the automatic emission of packets by the upper layer

### loss probability

0.2
the loss probability of every packet
Total Packets sent:

697

Total OK:

320

Useful BW (packets/s):

0.2923156607201744

Total BW (packets/s):

0.6367000485061299

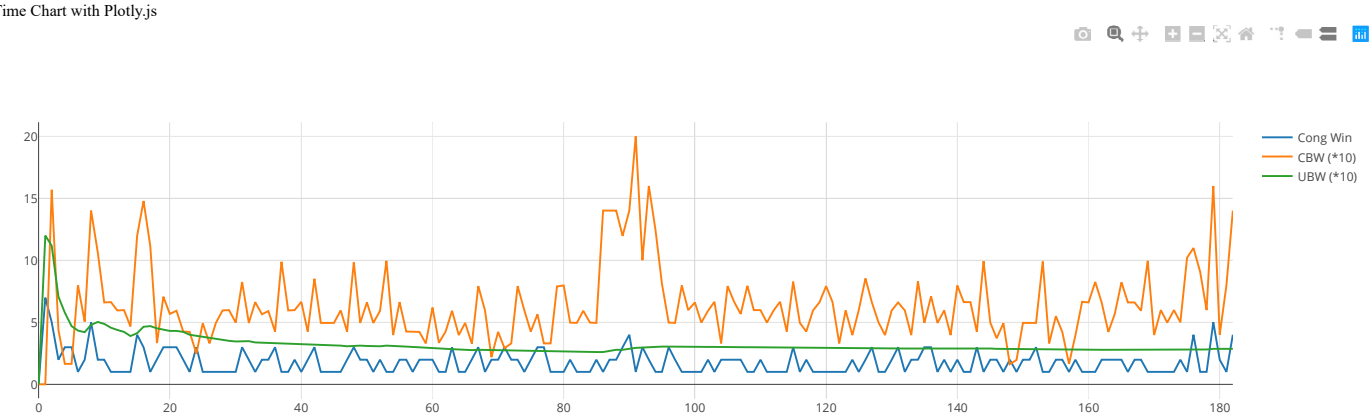
Current BW (packets/s):

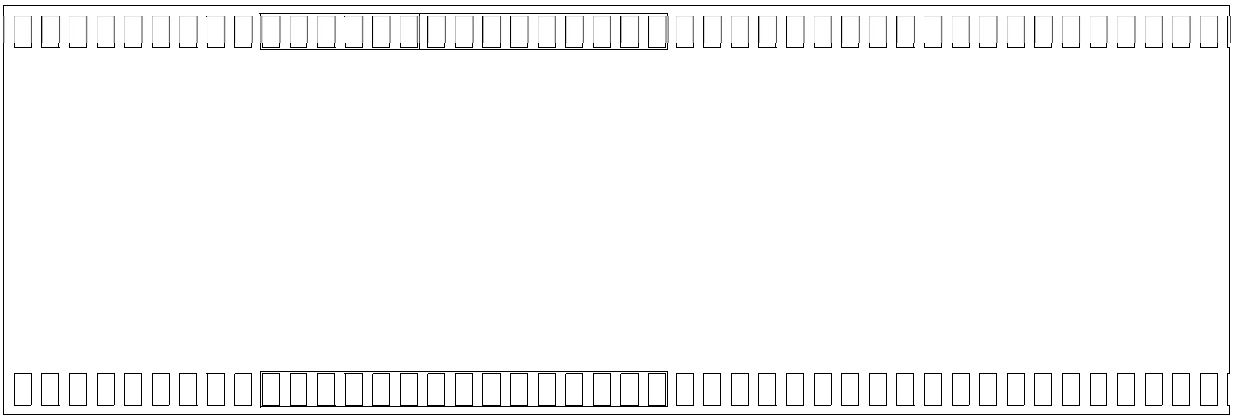
1.2432012432012434

Loss Prob:

0.2756574004507889

Real-Time Chart with Plotly.js





## legend

- no data received yet
- data buffered (ready to send, delivered or sent but no ack received yet)
- ack
- cumulative ack
- transmission confirmed
- data has been delivered to upper network layer

coded by Johannes Kessler 2012

small extensions by Jo Piquer 2019

# Selective Repeat / Go Back N

## configuration

### protocol

( ) Go back N ( ) Selective Repeat (•) Selective Repeat+CACK
choosing a new protocol restarts the simulation

### window size

15
sets the window size for the windows

### end to end delay

500
time a packet takes from one station to the other

### end to end delay variance (+/- err)

0
+/- error added to end to end delay

### timeout

1500

### scroll mode

Typewriter style
change the style the window scrolls

### number of packets emited per minute

120
the number of packets the upper layer tries to send per minute

### automatic emission of packets

start
starts or stops the automatic emission of packets by the upper layer

### loss probability

0.2
the loss probability of every packet
Total Packets sent:

697

Total OK:

338

Useful BW (packets/s):

0.3014274999977705

Total BW (packets/s):

0.6215827440782428

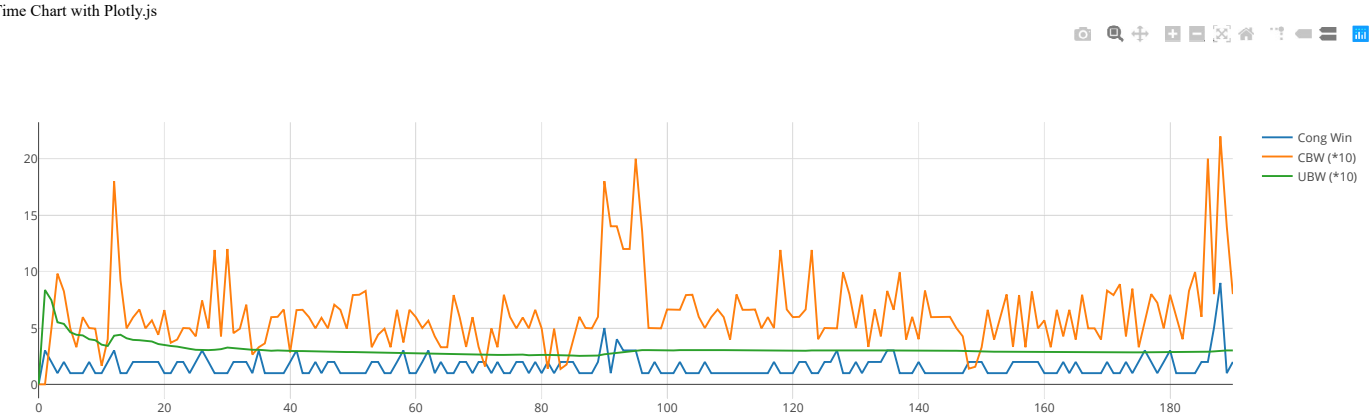
Current BW (packets/s):

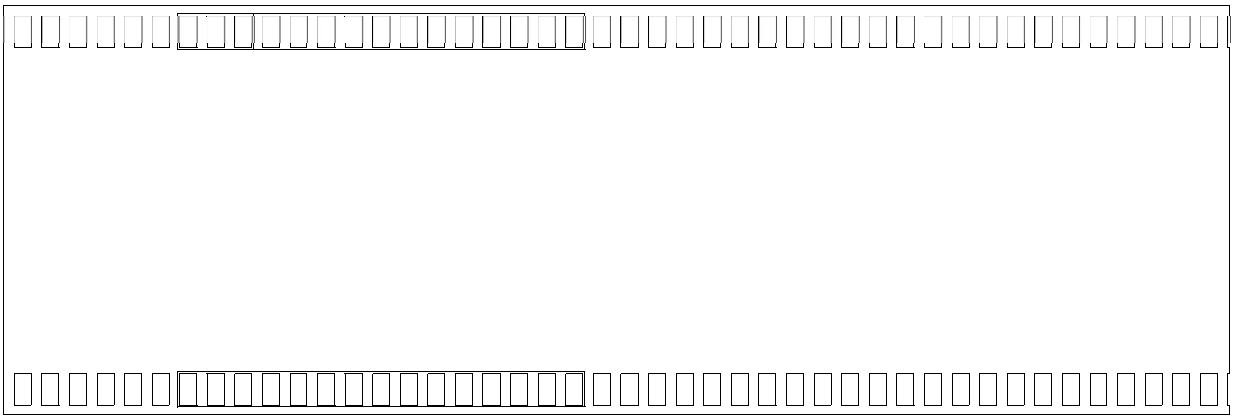
0.684404140645051

Loss Prob:

0.2

Real-Time Chart with Plotly.js





## legend

- no data received yet
- data buffered (ready to send, delivered or sent but no ack received yet)
- ack
- cumulative ack
- transmission confirmed
- data has been delivered to upper network layer

coded by Johannes Kessler 2012

small extensions by Jo Piquer 2019