



UNIVERSIDAD DE CHILE  
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS  
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

PRONÓSTICO DE LA TASA DE TRANSMISIÓN A NIVEL DE LA CAPA DE TRANSPORTE

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO

FELIPE GUILLERMO SALAS REYES

PROFESOR GUÍA:  
CLAUDIO ESTÉVEZ MONTERO

MIEMBROS DE LA COMISIÓN  
MARCOS ORCHARD CONCHA  
RODRIGO ARENAS ANDRADE

SANTIAGO DE CHILE  
2013

RESUMEN DE LA MEMORIA  
PARA OPTAR AL TÍTULO DE:  
INGENIERO CIVIL ELÉCTRICO  
POR: FELIPE SALAS REYES.  
FECHA: 04/10/2013  
PROF. GUÍA: DR. CLAUDIO ESTÉVEZ MONTERO

## “Pronóstico de la Tasa de Transmisión a Nivel de la Capa de Transporte”

Las telecomunicaciones han evolucionado a un nivel tal que prácticamente se volvieron parte de la cotidianidad de la sociedad. La cantidad de aplicaciones y servicios disponibles así como la facilidad de acceso para gran parte de la población, ya sea por los bajos costos o por incluso gratuidad de conexión en lugares específicos, hacen que los requisitos de capacidad de transferencia de datos crezcan por lo tanto un óptimo uso de la capacidad del canal de transmisión es una necesidad. Esta necesidad es el foco de este trabajo. En este trabajo se propone un algoritmo que aborda la problemática a nivel de la capa de transporte.

La capa de transporte utiliza el protocolo *TCP* para la transferencia segura e íntegra de los datos. Dado el funcionamiento de este protocolo, en donde se requiere confirmación de cada paquete recibido para la transmisión de datos (*ACK*), el *Round-Trip Time* (RTT) o retardo afecta drásticamente la velocidad de transmisión porque los paquetes en vuelo (que están siendo transferidos en el canal) retrasan el crecimiento de la ventana de congestión. Este problema se acentúa particularmente en las redes de *high Round-Trip Time* (BDP) que corresponden a redes que tienen una alta capacidad de transferencia y RTT altos provocando que la cantidad de datos en vuelo sea elevado. El uso de *TCP* en las redes de alto BDP hace que las transmisiones sean muy dependientes del retardo y su mejor utilización de recursos corresponde a un área de estudio muy activa dentro de los algoritmos de evasión de congestión.

En este trabajo se propone un método para encontrar la capacidad del canal de telecomunicaciones usando información que está disponible en la capa de transporte. El método consiste en utilizar una métrica basada en la distancia (secuencial o temporal) entre dos pérdidas y con esta información estimar la capacidad del canal. De esta forma se puede obtener un nivel de conocimiento sobre la red explícito en lugar de tantear mediante *probing*, método empleado por la mayoría de los protocolos existentes. Con la capacidad de canal estimada se calcula la ventana de congestión óptima teniendo varios factores en consideración como: *throughput*, *TCP-friendliness* y evasión de congestión. Además de conocer el valor de la capacidad del canal es necesario transmitir con ese conocimiento, por lo tanto también se propone la implementación de dos algoritmos de evasión de congestión en entorno a Linux. Las pruebas entregan resultados alentadores en donde se logra mantener la ventana de congestión muy cerca y establemente a la capacidad encontrada del canal. Esto se refiere principalmente a que las desviaciones estándar de la ventana de congestión de los módulos implementados tienen valores bajos y son alrededor de cuatro veces más pequeñas que las entregadas por el algoritmo Reno.

Como parte de la productividad de este estudio se lograron varios hitos. El resultado de este trabajo fue publicado en dos conferencias, Chilecon 2013 y IEEE Latincom 2013. Adicionalmente, estamos escribiendo un paper que esperamos publicar en el journal de IEEE/ACM Transactions on Networking. Por último, se considera que este trabajo tendrá un alto impacto en el área de investigación de protocolos de transporte a nivel mundial.

*“Gracias por el apoyo y el amor entregado, fueron las bases  
para lograr este fin”*

A Guillermo y Maritza, mis padres.

# AGRADECIMIENTOS

Al grupo de *Optical and Wireless Laboratory* por permitir el uso de sus equipos para la realización del proyecto.

A mi familia, Francisco, Guillermo y Maritza por su apoyo y afecto durante mi estancia en la universidad.

Al Doctor Claudio Estévez por darme la oportunidad de estar trabajando en esta área desde la práctica profesional. Por sus sugerencias, ideas y ayuda en la realización de la memoria.

A los profesores miembros de la comisión los señores Marcos Orchard y Rodrigo Arenas por aceptar esta labor y por sus observaciones del presente documento.

A mis amigos de la universidad que hicieron que desde primer año esta etapa de mi vida fuese más divertida con tantos ratos de estudio grupal ineficiente, de taca-taca y de asados.

# TABLA DE CONTENIDO

<b>AGRADECIMIENTOS.....</b>	<b>IV</b>
<b>CAPÍTULO 1:.....</b>	<b>INTRODUCCIÓN 1</b>
1.1	MOTIVACIÓN ..... 1
1.2	OBJETIVOS..... 3
1.2.1	<i>Objetivos generales</i> ..... 3
1.2.2	<i>Objetivos específicos</i> ..... 3
1.3	ESTRUCTURA..... 4
<b>CAPÍTULO 2:.....</b>	<b>CONTEXTUALIZACIÓN Y ESTADO DEL ARTE 5</b>
2.1	INTRODUCCIÓN..... 5
2.2	MODELO OSI..... 5
2.3	CAPA DE TRANSPORTE..... 8
2.4	PROTOCOLO IP..... 8
2.4.1	<i>Comparación de conmutación de paquetes de Redes IP con redes de conmutación de circuitos</i> ..... 9
2.5	TCP ..... 9
2.5.1	<i>Funcionamiento de TCP en la transferencia de datos</i> ..... 9
2.6	ALGORITMOS PARA EVITAR CONGESTIÓN..... 10
2.6.1	<i>Algunos Algoritmos de control de congestión utilizados:</i> ..... 12
2.7	ESTIMACIÓN TEÓRICA DEL <i>THROUGHPUT</i> RENO ..... 15
2.8	MODELO AIMD ..... 16
2.9	HIGH BANDWIDTH DELAY PRODUCT NETWORKS..... 17
2.10	ESTIMACIÓN DE <i>THROUGHPUT</i> COMO HERRAMIENTA..... 19
<b>CAPÍTULO 3:.....</b>	<b>IMPLEMENTACIÓN 20</b>
3.1	INTRODUCCIÓN..... 20
3.2	PREDICCIÓN DE LA VENTANA DE CONGESTIÓN ..... 20
3.3	CONVERGENCIA A LA CAPACIDAD DEL CANAL..... 23
3.3.1	<i>Método de diferencia porcentual</i> ..... 23
3.3.2	<i>Método usando un controlador PI</i> ..... 24
3.4	SIMULACIÓN OPNET ..... 25
3.5	IMPLEMENTACIÓN ENTORNO LINUX ..... 29
3.5.1	<i>Entorno de pruebas</i> ..... 29
3.5.2	<i>Pasos previos</i> ..... 30
3.5.3	<i>Estructura de TCP en Linux</i> ..... 33
3.5.4	<i>Software requerido</i> ..... 36
3.5.5	<i>Procedimiento de operación del modulo</i> ..... 40
<b>CAPÍTULO 4:.....</b>	<b>ANÁLISIS DE RESULTADOS 42</b>
4.1	INTRODUCCIÓN..... 42
4.2	RESULTADOS DE RTT VARIABLE..... 43
4.2.1	<i>Módulo tcp_conn_alt</i> ..... 44
4.2.2	<i>Módulo tcp_conn_pi</i> ..... 50
4.2.3	<i>Módulo Reno para análisis comparativo</i> ..... 56
4.3	RESULTADOS DE TASA DE PÉRDIDA VARIABLE..... 60

4.3.1	Módulo <i>tcp_conn_alt</i> .....	61
4.3.2	Módulo <i>tcp_conn_pi</i> .....	67
4.3.3	Módulo <i>Reno para análisis comparativo</i> .....	74
4.4	RESULTADOS CON VALORES DEL CONTROLADOR PI VARIABLES .....	77
4.5	RESULTADOS CON VALOR DE PESO VARIABLE Y TIPO DE PESO .....	84
4.5.1	<i>Peso del tipo <math>W \sim 1n</math></i> .....	85
4.5.2	<i>Peso constante</i> .....	88
4.6	RESULTADOS CON CAMBIOS EN LA CAPACIDAD DEL CANAL DURANTE LA TRANSMISIÓN.....	92
4.7	TABLA RESUMEN DE ESTADÍSTICAS .....	96
4.7.1	<i>RTT variable</i> .....	96
4.7.2	<i>Tasa de pérdida variable</i> .....	98
<b>CAPÍTULO 5: .....</b>		<b>CONCLUSIONES Y TRABAJO FUTURO 100</b>
5.1	INTRODUCCIÓN.....	100
5.2	CONCLUSIONES.....	100
5.3	TRABAJO FUTURO.....	102
<b>CAPÍTULO 6: .....</b>		<b>BIBLIOGRAFÍA 103</b>
<b>ANEXOS .....</b>		<b>105</b>
CÓDIGO TCP_CONN_ALT EN OPNET PARA SIMULACIONES.....		105
<i>Desde Línea 973</i> .....		105
<i>Desde línea 2066</i> .....		106
CÓDIGO TCP_CONN_PI EN OPNET PARA SIMULACIONES .....		107
<i>Desde línea 951</i> .....		107
<i>Desde la línea 2112</i> .....		109
CÓDIGO TCP_RENO UTILIZADO CON INCLUSIÓN DEL CÁLCULO DE LA CAPACIDAD DEL CANAL.....		110
CÓDIGO DEL MÓDULO TCP_CONN_ALT EN LINUX .....		113
CÓDIGO DEL MÓDULO TCP_CONN_PI EN LINUX.....		118
CÓDIGO DEL BASH OPCIONES_BUFF .....		123
CÓDIGO DE SCRIPT LECTOR DE ARCHIVOS TCP_PROBE EN MATLAB .....		126

# ÍNDICE DE ILUSTRACIONES

ILUSTRACIÓN 1: CAPAS DEL MODELO OSI.....	6
ILUSTRACIÓN 2: ESQUEMA DE ENCAPSULAMIENTO DE ACUERDO A LAS CAPAS DEL MODELO OSI.....	6
ILUSTRACIÓN 3: COMUNICACIÓN CON NODOS INTERMEDIOS.....	7
ILUSTRACIÓN 4: ESQUEMA DE TRANSFERENCIA TCP .....	10
ILUSTRACIÓN 5: GRÁFICA DE EJEMPLO DEL COMPORTAMIENTO DE TCP RENO .....	12
ILUSTRACIÓN 6: GRÁFICA DE EJEMPLO DEL COMPORTAMIENTO DE TCP VEGAS.....	13
ILUSTRACIÓN 7: GRÁFICA DE EJEMPLO DEL COMPORTAMIENTO DE TCP BIC.....	14
ILUSTRACIÓN 8: GRÁFICA DE EJEMPLO DEL COMPORTAMIENTO DE TCP CUBIC .....	15
ILUSTRACIÓN 9: DIAGRAMA DE BLOQUES DE UN CONTROLADOR PROPORCIONAL.....	17
ILUSTRACIÓN 10: DIAGRAMA DE BLOQUES DE UN CONTROLADOR INTEGRAL.....	18
ILUSTRACIÓN 11: DIAGRAMA DE BLOQUES DE UN CONTROLADOR PI.....	18
ILUSTRACIÓN 12: VENTANA DE CONGESTIÓN DE RENO, VENTANA DE CONGESTIÓN PREDICHA Y VALOR ESPERADO DE LA VENTANA DE CONGESTIÓN .....	21
ILUSTRACIÓN 13: $\Delta$ COMO DISTANCIA ENTRE DOS PÉRDIDAS. ....	21
ILUSTRACIÓN 14: ROJO: VENTANA DE CONGESTIÓN DE RENO. AZUL: VENTANA PREDICHA.....	22
ILUSTRACIÓN 15: ESQUEMA DE CONTROL PARA EL SISTEMA .....	24
ILUSTRACIÓN 16: NODO FORZADO DE COLOR VERDE Y NODO NO FORZADO DE COLOR ROJO .....	26
ILUSTRACIÓN 17: ESCENARIO OPNET DE SIMULACIÓN DE RENO .....	26
ILUSTRACIÓN 18: SIMULACIÓN USANDO ALGORITMO RENO, PÉRDIDA DE 0.1% Y RTT DE 10[ms] .....	26
ILUSTRACIÓN 19: MODELO NODAL DEL SERVIDOR.....	27
ILUSTRACIÓN 20: MODELO NODAL DE TCP .....	27
ILUSTRACIÓN 21: MODELO NODAL DE LOS <i>CHILD PROCESS</i> .....	28
ILUSTRACIÓN 22: VENTANA PREDICHA ENCONTRADA PARA ALGORITMO RENO EN LA SIMULACIÓN.....	28
ILUSTRACIÓN 23: ESQUEMA DE CONEXIÓN DEL ENTORNO DE PRUEBAS.....	29
ILUSTRACIÓN 24: FUENTES Y CABECERAS DEL KERNEL DE LINUX OBTENIDAS CON SYNAPTIC .....	30
ILUSTRACIÓN 25: MODULE-ASSISTANT PARA DEJAR AL SISTEMA OPERATIVO CON LA CAPACIDAD DE COMPILAR MÓDULOS. ....	31
ILUSTRACIÓN 26: MÓDULO TCP_CONN_ALT RTT=10[ms] Y LOSS=0.1%.....	44
ILUSTRACIÓN 27: MÓDULO TCP_CONN_ALT RTT=50[ms] Y LOSS=0.1%.....	44
ILUSTRACIÓN 28: MÓDULO TCP_CONN_ALT RTT=100[ms] Y LOSS=0.1%.....	45
ILUSTRACIÓN 29: MÓDULO TCP_CONN_ALT RTT=10[ms] Y LOSS=0.001%.....	45
ILUSTRACIÓN 30: MÓDULO TCP_CONN_ALT RTT=50[ms] Y LOSS=0.001%.....	46
ILUSTRACIÓN 31: MÓDULO TCP_CONN_ALT RTT=100[ms] Y LOSS=0.001%.....	46
ILUSTRACIÓN 32: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_ALT PARA RTT VARIABLE CON LOSS=0.1%.....	47
ILUSTRACIÓN 33: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_ALT PARA RTT VARIABLE CON LOSS=0.1%.....	48
ILUSTRACIÓN 34: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_ALT PARA RTT VARIABLE CON LOSS=0.001%.....	48
ILUSTRACIÓN 35: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_ALT PARA RTT VARIABLE CON LOSS=0.001%.....	49
ILUSTRACIÓN 36: MÓDULO TCP_CONN_PI RTT=40[ms] Y LOSS=0.1% .....	50
ILUSTRACIÓN 37: MÓDULO TCP_CONN_PI RTT=50[ms] Y LOSS=0.1% .....	51
ILUSTRACIÓN 38: MÓDULO TCP_CONN_PI RTT=100[ms] Y LOSS=0.1% .....	51
ILUSTRACIÓN 39: MÓDULO TCP_CONN_PI RTT=10[ms] Y LOSS=0.001% .....	52
ILUSTRACIÓN 40: MÓDULO TCP_CONN_PI RTT=50[ms] Y LOSS=0.001% .....	52
ILUSTRACIÓN 41: MÓDULO TCP_CONN_PI RTT=100[ms] Y LOSS=0.001%.....	53
ILUSTRACIÓN 42: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA RTT VARIABLE CON LOSS=0.1% .....	54
ILUSTRACIÓN 43: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA RTT VARIABLE CON LOSS=0.1%.....	54
ILUSTRACIÓN 44: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA RTT VARIABLE CON LOSS=0.001% .....	55
ILUSTRACIÓN 45: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA RTT VARIABLE CON LOSS=0.001%.....	55
ILUSTRACIÓN 46: MÓDULO TCP_RENO RTT=100[ms] Y LOSS=0.1%.....	57

ILUSTRACIÓN 47: MÓDULO TCP_RENO RTT=100[ms] y Loss=0.001% .....	57
ILUSTRACIÓN 48: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_RENO PARA RTT VARIABLE CON Loss=0.1% .....	58
ILUSTRACIÓN 49: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_RENO PARA RTT VARIABLE CON Loss=0.1% ....	58
ILUSTRACIÓN 50: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_RENO PARA RTT VARIABLE CON Loss=0.001% .....	59
ILUSTRACIÓN 51: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_RENO PARA RTT VARIABLE CON Loss=0.001% .....	59
ILUSTRACIÓN 52: MÓDULO TCP_CONN_ALT RTT=10[ms] y Loss=0.001% .....	61
ILUSTRACIÓN 53: MÓDULO TCP_CONN_ALT RTT=10[ms] y Loss=0.01% .....	62
ILUSTRACIÓN 54: MÓDULO TCP_CONN_ALT RTT=10[ms] y Loss=0.1% .....	62
ILUSTRACIÓN 55: MÓDULO TCP_CONN_ALT RTT=100[ms] y Loss=0.001% .....	63
ILUSTRACIÓN 56: MÓDULO TCP_CONN_ALT RTT=100[ms] y Loss=0.01% .....	63
ILUSTRACIÓN 57: MÓDULO TCP_CONN_ALT RTT=100[ms] y Loss=0.1% .....	64
ILUSTRACIÓN 58: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_ALT PARA TASA DE PÉRDIDA VARIABLE CON RTT=10[ms]	65
ILUSTRACIÓN 59: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_ALT PARA TASA DE PÉRDIDA VARIABLE CON RTT=10[ms] .....	65
ILUSTRACIÓN 60: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_ALT PARA TASA DE PÉRDIDA VARIABLE CON RTT=100[ms]	66
ILUSTRACIÓN 61: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_ALT PARA TASA DE PÉRDIDA VARIABLE CON RTT=100[ms] .....	66
ILUSTRACIÓN 62: MÓDULO TCP_CONN_PI RTT=10[ms] y Loss=0.001% .....	68
ILUSTRACIÓN 63: MÓDULO TCP_CONN_PI RTT=10[ms] y Loss=0.01% .....	68
ILUSTRACIÓN 64: MÓDULO TCP_CONN_PI RTT=10[ms] y Loss=0.1% .....	69
ILUSTRACIÓN 65: MÓDULO TCP_CONN_PI RTT=100[ms] y Loss=0.001% .....	69
ILUSTRACIÓN 66: MÓDULO TCP_CONN_PI RTT=100[ms] y Loss=0.01% .....	70
ILUSTRACIÓN 67: MÓDULO TCP_CONN_PI RTT=100[ms] y Loss=0.1% .....	70
ILUSTRACIÓN 68: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA TASA DE PÉRDIDA VARIABLE CON RTT=10[ms] ..	71
ILUSTRACIÓN 69: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA TASA DE PÉRDIDA VARIABLE CON RTT=10[ms] .....	72
ILUSTRACIÓN 70: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA TASA DE PÉRDIDA VARIABLE CON RTT=100[ms]	72
ILUSTRACIÓN 71: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA TASA DE PÉRDIDA VARIABLE CON RTT=100[ms] .....	73
ILUSTRACIÓN 72: MÓDULO TCP_RENO RTT=10[ms] y Loss=0.1% .....	74
ILUSTRACIÓN 73: MÓDULO TCP_RENO RTT=100[ms] y Loss=0.1% .....	75
ILUSTRACIÓN 74: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_RENO PARA TASA DE PÉRDIDA VARIABLE CON RTT=10[ms] .....	75
ILUSTRACIÓN 75: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_RENO PARA TASA DE PÉRDIDA VARIABLE CON RTT=10[ms] .....	76
ILUSTRACIÓN 76: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_RENO PARA TASA DE PÉRDIDA VARIABLE CON RTT=100[ms] .....	76
ILUSTRACIÓN 77: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_RENO PARA TASA DE PÉRDIDA VARIABLE CON RTT=100[ms] .....	77
ILUSTRACIÓN 78: MÓDULO TCP_CONN_PI RTT=10[ms], Loss=0.1%, $K_p=10^{-5}$ , R=100 .....	78
ILUSTRACIÓN 79: MÓDULO TCP_CONN_PI RTT=10[ms], Loss=0.1%, $K_p=5 \cdot 10^{-5}$ , R=100 .....	79
ILUSTRACIÓN 80: MÓDULO TCP_CONN_PI RTT=10[ms], Loss=0.1%, $K_p=10^{-4}$ , R=100 .....	79
ILUSTRACIÓN 81: MÓDULO TCP_CONN_PI RTT=10[ms], Loss=0.1%, $K_p=5 \cdot 10^{-4}$ , R=100 .....	80
ILUSTRACIÓN 82: MÓDULO TCP_CONN_PI RTT=10[ms], Loss=0.1%, $K_p=10^{-5}$ , R=1000 .....	80
ILUSTRACIÓN 83: MÓDULO TCP_CONN_PI RTT=10[ms], Loss=0.1%, $K_p=5 \cdot 10^{-5}$ , R=1000 .....	81
ILUSTRACIÓN 84: MÓDULO TCP_CONN_PI RTT=10[ms], Loss=0.1%, $K_p=10^{-4}$ , R=1000 .....	81
ILUSTRACIÓN 85: MÓDULO TCP_CONN_PI RTT=10[ms], Loss=0.1%, $K_p=5 \cdot 10^{-4}$ , R=1000 .....	82
ILUSTRACIÓN 86: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA K VARIABLE CON TASA DE PÉRDIDA=0.1%, RTT=10[ms] y R=100 .....	82
ILUSTRACIÓN 87: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA K VARIABLE CON TASA DE PÉRDIDA=0.1%, RTT=10[ms] y R=100 .....	83
ILUSTRACIÓN 88: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA K VARIABLE CON TASA DE PÉRDIDA=0.1%, RTT=10[ms] y R=1000 .....	83



ILUSTRACIÓN 89: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TCP_CONN_PI PARA K VARIABLE CON TASA DE PÉRDIDA=0.1%, RTT=10[ms] Y R=1000 .....	84
ILUSTRACIÓN 90: CWND_PRED MÓDULO TCP_CONN_CONN CON PESO W=1 Y RTT=10[ms] Y LOSS=0.1% .....	85
ILUSTRACIÓN 91: CWND_PRED MÓDULO TCP_CONN_CONN CON PESO W=5 Y RTT=10[ms] Y LOSS=0.1% .....	86
ILUSTRACIÓN 92: CWND_PRED MÓDULO TCP_CONN_CONN CON PESO W=10 Y RTT=10[ms] Y LOSS=0.1% .....	86
ILUSTRACIÓN 93: MEDIA DE LA VENTANA DE CONGESTIÓN PREDICHA PARA TCP_CONN_ALT PARA PESO W VARIABLE CON TASA DE PÉRDIDA=0.1% Y RTT=10[ms] .....	87
ILUSTRACIÓN 94: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PREDICHA PARA TCP_CONN_ALT PARA PESO W VARIABLE CON TASA DE PÉRDIDA=0.1% Y RTT=10[ms] .....	87
ILUSTRACIÓN 95: MÓDULO TCP_CONN_PI RTT=10[ms], LOSS=0.1% Y PESO PROPORCIONAL W=1% .....	88
ILUSTRACIÓN 96: MÓDULO TCP_CONN_PI RTT=10[ms], LOSS=0.1% Y PESO PROPORCIONAL W=5% .....	89
ILUSTRACIÓN 97: MÓDULO TCP_CONN_PI RTT=10[ms], LOSS=0.1% Y PESO PROPORCIONAL W=10% .....	89
ILUSTRACIÓN 98: MEDIA DE LA VENTANA DE CONGESTIÓN PREDICHA PARA TCP_CONN_PI PARA PESO W PROPORCIONAL VARIABLE CON TASA DE PÉRDIDA=0.1% Y RTT=10[ms] .....	90
ILUSTRACIÓN 99: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PREDICHA PARA TCP_CONN_PI PARA PESO W PROPORCIONAL VARIABLE CON TASA DE PÉRDIDA=0.1% Y RTT=10[ms] .....	91
ILUSTRACIÓN 100: MÓDULO TCP_CONN_PI RTT=10[ms] CON CAPACIDAD DEL CANAL CAMBIANTE DE LOSS=0.1% A 0.001% Y PESO W=1 .....	92
ILUSTRACIÓN 101: MÓDULO TCP_CONN_PI RTT=10[ms] CON CAPACIDAD DEL CANAL CAMBIANTE DE LOSS=0.1% A 0.001% Y PESO W=10 .....	93
ILUSTRACIÓN 102: MÓDULO TCP_CONN_PI RTT=10[ms] CON CAPACIDAD DEL CANAL CAMBIANTE DE LOSS=0.1% A 0.001% Y PESO PROPORCIONAL W=5% .....	93
ILUSTRACIÓN 103: MÓDULO TCP_RENO RTT=10[ms] CON CAPACIDAD DEL CANAL CAMBIANTE DE LOSS=0.1% A 0.001% Y PESO W=194 .....	93

## ÍNDICE DE TABLAS

TABLA 1: COMPARACION ENTRE CONMUTACION DE CIRCUITOS VERSUS CONMUTACIÓN DE PAQUETES .....	9
TABLA 2: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TASA DE PÉRDIDA DE 0.1% .....	96
TABLA 3: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TASA DE PÉRDIDA DE 0.1% .....	96
TABLA 4: MEDIA DE LA VENTANA DE CONGESTIÓN PARA TASA DE PÉRDIDA DE 0.001% .....	97
TABLA 5: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA TASA DE PÉRDIDA DE 0.001% .....	97
TABLA 6: MEDIA DE LA VENTANA DE CONGESTIÓN PARA RTT=10[ms] .....	98
TABLA 7: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA RTT=10[ms] .....	98
TABLA 8: MEDIA DE LA VENTANA DE CONGESTIÓN PARA RTT=100[ms] .....	98
TABLA 9: DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA RTT=100[ms] .....	99
TABLA 10: MEDIA Y DESVIACIÓN ESTÁNDAR DE LA VENTANA DE CONGESTIÓN PARA KP VARIABLE .....	99
TABLA 11: MEDIA Y DESVIACIÓN ESTÁNDAR DE LA VENTANA PREDICHA PARA PESOS VARIABLES .....	99

# LISTA DE ACRÓNIMOS

AIMD	<i>Additive increase multiplicative decrease</i>
ACK	<i>Acknowledgement</i>
BDP	<i>Bandwidth-delay product</i>
CIR	<i>Committed information rate</i>
cwnd	<i>Congestion window</i>
EIR	<i>Excess Information Rate</i>
NetEm	<i>Network Emulator</i>
OSI	<i>Open system interconnection</i>
RTT	<i>Round-Trip Time</i>
TCP	<i>Transmission Control Protocol</i>

# CAPÍTULO 1: INTRODUCCIÓN

## 1.1 Motivación

Las telecomunicaciones son un pilar fundamental en el desarrollo del mundo moderno ya que la cantidad de servicios y aplicaciones ofrecidas es altísima y se van incrementando de acuerdo a las necesidades de la sociedad. Los dispositivos terminales ya no corresponden únicamente a grandes computadores sino más bien se tratan de dispositivos pequeños y móviles tales como teléfonos inteligentes, *netbooks*, *tablets* y otros aparatos terminales. De manera que prácticamente los individuos tienen la posibilidad de estar siempre conectados. Este cambio en donde un equipo con conexión en promedio por casa cambia a un equipo por persona claramente genera un aumento en el tráfico de datos de la red. Sin contar que las propias aplicaciones consumen más recursos acorde al avance del tiempo, primero se transmitía texto, luego imágenes de cierta resolución, luego videos.

*Transmission Control Protocol (TCP)* corresponde al protocolo de la capa de transporte de Internet orientado a la conexión, su función es mantener la comunicación fiable, libre de errores y sin pérdida (datos enviados son los mismos datos recibidos). Para lograr esto se *TCP* utiliza acuses de recibo y control de congestión.

Un caso particular importante se trata de las redes *high bandwidth delay product* en donde el producto del retardo entre los equipos intercomunicados y la tasa de transmisión es muy alta. Este tipo de red tiene la característica de tener una gran cantidad de datos en vuelo, es decir, en el propio canal y por ello requiere un diseño de protocolo especial, como en *TCP* en referencia a la optimización del rendimiento. Se propone la implementación de un protocolo de evasión de congestión para *TCP* cuya finalidad sea obtener la capacidad del canal de transmisión.

En *TCP* existen diversos algoritmos de evasión de congestión que poseen dos funciones principales: evitar que el sistema llegue a un estado de congestión y en caso de llegar a dicha situación volver al sistema a su estado normal. Estos protocolos pueden ser orientados al tipo de red en particular: *wifi*, satelital, *high bandwidth delay product*, etc. Sin embargo, no existe un protocolo óptimo por un tema de complejidad del sistema y se trabaja en un nivel subóptimo.

Dado el funcionamiento del protocolo *TCP* en donde este requiere confirmación de cada paquete para la transmisión de datos (*ACK*), el *Round-Trip Time (RTT)*, tiempo que tarda un paquete enviado por un emisor en volver al mismo emisor una vez pasado por el receptor) afecta drásticamente la velocidad de transmisión porque los paquetes en vuelo tardan bastante en desplazarse. El primer problema corresponde a que la ventana de congestión oscila por

debajo de la capacidad del canal y por naturaleza alcanza tres cuartos de la capacidad máxima cuando existe un *EIR* (*excess information rate*), es decir, un control de tráfico (que es bastante común). Además hay un segundo problema en que para las ya planteadas redes *high bandwidth delay product* el rendimiento es mucho peor.

Los protocolos de evasión de congestión de la capa de transporte en *TCP* son controlados por el servidor, es decir, aquel terminal que envía los datos. Ya que el servidor va a tratar de enviar los datos a la capacidad que el cliente los pueda recibir, caso contrario resultaría en un protocolo no orientado a la conexión y por lo tanto no tendría sentido usar *TCP*.

Obtener la capacidad del canal de transmisión puede ayudar en parte a un mejor diseño de un protocolo de evasión de congestión en cuanto a una utilización más eficiente de los recursos de la red, pues usualmente se trabaja mediante *probing*, es decir, mediante una exploración de los posibles valores que toma la ventana de congestión y en base a una métrica comenzar a regularse. El presente trabajo de memoria tiene como objetivo obtener una manera de encontrar la capacidad del canal de acuerdo a alguna métrica y usarla en la implementación de dos algoritmos de evasión de congestión y evaluar su rendimiento respecto entre ellos mismos y el protocolo Reno que corresponde a uno de los más comunes utilizados.

Lo ideal en cuanto al funcionamiento de los protocolos a implementar difiere de un *probing* pues al tener la capacidad del canal como dato conocido se intenta dejar la tasa de transferencia en dicho valor o lo más cercano a él durante el mayor tiempo posible utilizando un controlador proporcional porcentual y un controlador PI (proporcional integral, es decir, tanto el error como la integral de éste modifican el comportamiento del controlador). Para que sea esto sea factible es necesario que la forma en la cual se define la manera de encontrar la capacidad del canal sea correcta pues de lo contrario los protocolos a implementar carecerían de valor. En términos simples la forma de encontrar la capacidad del canal utiliza la métrica de distancia entre dos paquetes perdidos, lo que es equivalente al recíproco de la pérdida instantánea, por lo tanto el nivel de pérdida de paquetes corresponde a la métrica para calcular la capacidad. Esto solo es válido cuando la pérdida y la congestión tienen una relación directa, lo cual es válido para redes *high bandwidth delay product* en donde la pérdida no tiende a ser demasiado alta.

Se pretende implementar los protocolos en el sistema operativo Linux por dos motivos: ser un sistema de código abierto en donde cualquier usuario con los conocimientos necesarios pueda modificar lo que desee, y segundo porque los protocolos de congestión en este sistema operativo son módulos conectables (*pluggable modules*), es decir, no se requiere una compilación del *kernel* completo cuando se altera algunos de los protocolos, sino que únicamente se compilan aquellos que se crean o modifican.

Cabe destacar además que las dos partes constituyentes del proyecto: encontrar la capacidad del canal y la implementación de los módulos de evasión de congestión en Linux, solo ésta última depende de la primera, por lo tanto es posible utilizar la capacidad del canal para otros futuros y posibles proyectos. Por ejemplo utilizar la capacidad como referencia para algún otro tipo de controlador más inteligente y eficiente o también para mejorar algún protocolo de congestión ya existente.

## 1.2 Objetivos

Los objetivos generales y específicos se entregan a continuación:

### 1.2.1 Objetivos generales

Los objetivos generales son encontrar la capacidad del canal e implementar los algoritmos para evitar congestión que converjan a la capacidad de la red en ese instante. Esto porque en el primer caso la existencia de un EIR no sería problema y sin congestión el valor límite de *throughput* impuesto por el EIR sería el valor de convergencia, y en el segundo caso sabiendo la capacidad de transmisión, se realiza un proceso de ingeniería inversa para determinar la cantidad de paquetes a enviar por la red *high bandwidth delay product*.

### 1.2.2 Objetivos específicos

Los objetivos específicos del proyecto corresponden:

- Encontrar la capacidad del canal y verificar el resultado con la teoría.
- Dejar al equipo servidor Linux con el módulo de evasión de congestión propuesto operativo.
- El modulo debe cumplir ciertos parámetros de diseño de acuerdo a la características del protocolo *TCP* (fidelidad, *fairness*, *TCP-friendly*)
- Verificar el funcionamiento del módulo mediante diversas pruebas.

## 1.3 Estructura

La memoria está estructurada en cinco capítulos distribuidos de la siguiente manera partiendo por la introducción ya mostrada:

El Capítulo 2 consiste en la presentación del estado del arte relacionado principalmente con *TCP* y sus respectivos protocolos de control de congestión y su funcionamiento, además de los conceptos básicos necesarios para el correcto entendimiento del presente proyecto de memoria.

El Capítulo 3 corresponde a la implementación y va desde la obtención de la capacidad del canal hasta la implementación del protocolo en el entorno Linux pasando por todos los software necesarios para dicho fin y para las pruebas a realizar. Este capítulo está expuesto de tal manera para que el lector pueda repetir el proyecto.

El Capítulo 4 entrega los resultados obtenidos de las pruebas realizadas de los módulos implementados para los dos tipos de controladores, diferentes condiciones de red, etc. Se verifica también la certeza del valor de la capacidad del canal de acuerdo a ciertos parámetros para aseverar que está dentro de los rangos esperados.

El Capítulo 5 corresponde a las conclusiones del proyecto respecto a los resultados obtenidos en el capítulo previo.

# CAPÍTULO 2: CONTEXTUALIZACIÓN Y ESTADO DEL ARTE

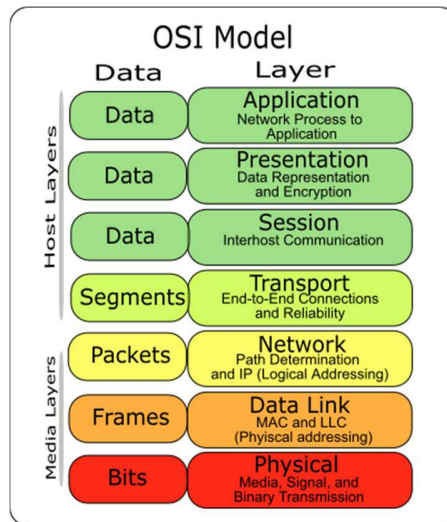
## 2.1 Introducción

Dado el alcance de la memoria es necesario realizar una explicación del funcionamiento del protocolo *TCP* que corresponde a la capa de transporte del modelo *open system interconnection* (OSI) y de los algoritmos para evitar congestión presente en dicho protocolo. Para ello se parte con una breve descripción del modelo OSI, mostrando las cualidades de una capa de transporte genérica, luego se incluye un breve acercamiento al protocolo IP como introducción a *TCP* y los algoritmos para evitar congestión. Esto porque estos últimos corresponden a la herramienta principal para implementar y desarrollar los objetivos de la memoria. Existe una sección que introduce el controlador PI que se utilizara dentro del módulo de evasión de congestión. También se explyaya respecto a las redes *high bandwidth delay product*, pues éstas corresponden a redes de mucha capacidad de transferencia y a la vez poseen un tiempo de transmisión de paquetes elevado. En este tipo de redes es donde resultan más aplicables los resultados de proyecto pues como se mostrara más adelante corresponde a un área en donde hay deficiencias en cuanto al uso del canal.

## 2.2 Modelo OSI

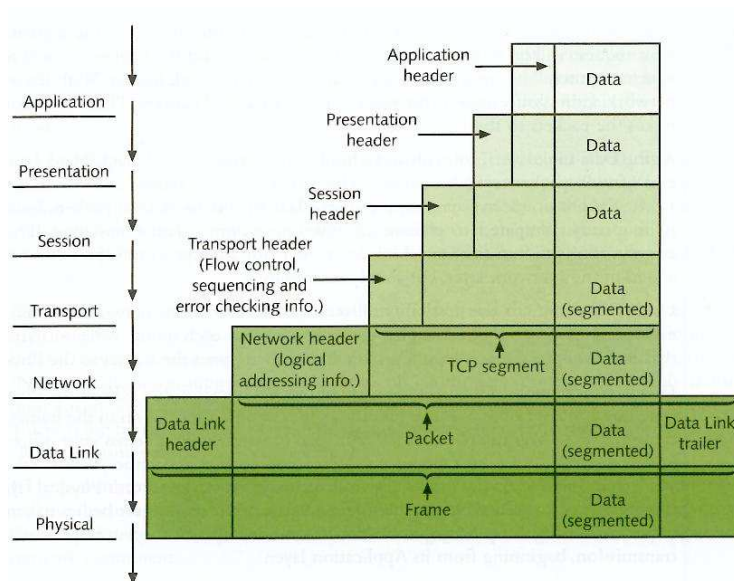
El modelo OSI [6] corresponde a un modelo de red, creado por la ISO, para ser tomado como referencia definiendo siete capas que corresponden a las distintas etapas necesarias para realizar la transmisión de información por un canal de telecomunicaciones. Además se trata de una descripción abstracta de los procesos de la comunicación.

Las capas corresponden en orden descendiente: capa de aplicación; capa de presentación; capa de sesión; capa de transporte; capa de red; capa de enlace de datos; capa física.



**Ilustración 1: Capas del modelo OSI**

Una de las principales características de este modelo corresponde a que la comunicación entre distintos dispositivos es capa a capa. Es decir lo que cada capa quiere dar a entender solo se entenderá por la misma capa del otro terminal operante de la comunicación. Esto es porque cada capa va agregando una cabecera a la información de las capas superiores en cuanto a la transmisión, y en lo referente a la recepción se van extrayendo las cabeceras de la información procedente de las capas inferiores.



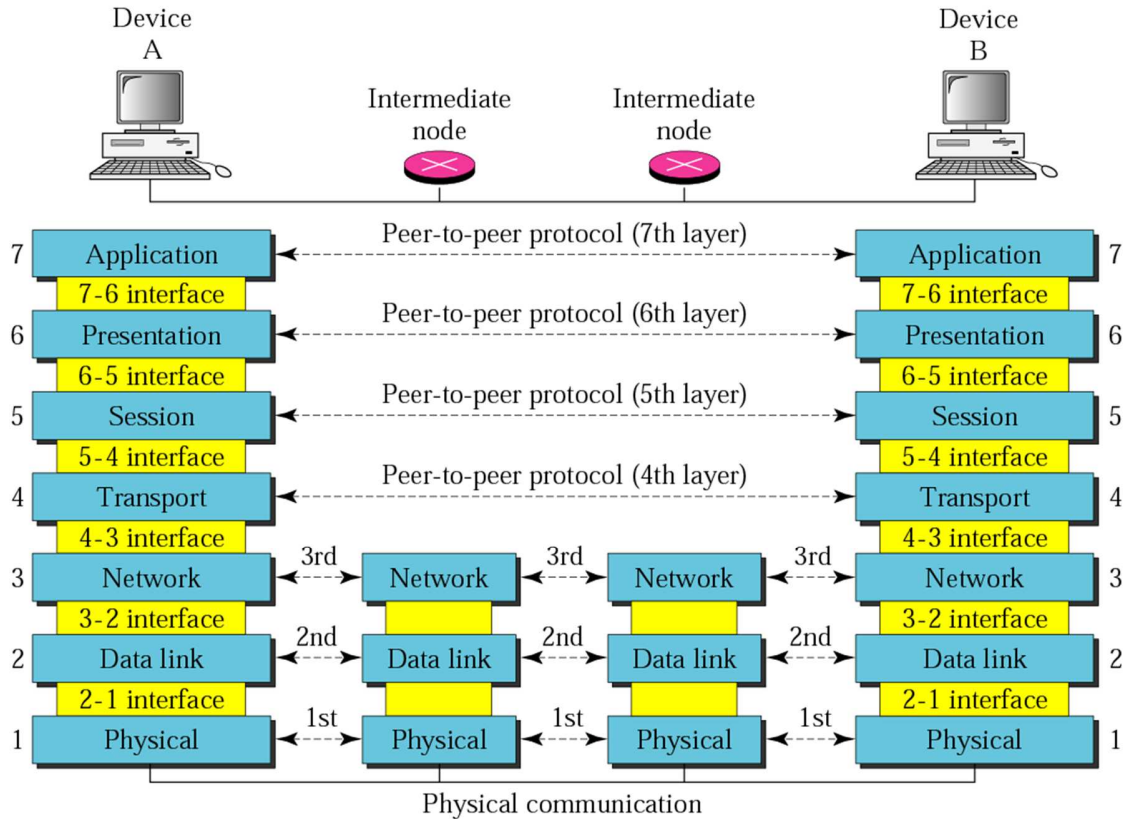
**Ilustración 2: Esquema de encapsulamiento de acuerdo a las capas del modelo OSI**

La idea de crear un modelo en base a capas se debe a que cada una de estas le agrega valor a los servicios proveídos por el conjunto de las capas inferiores, de esta manera la capa



más alta tiene a su disposición toda la gama de servicios necesarios que las aplicaciones requieran para su correcto funcionamiento.

Durante una conexión en la que exista algún nodo entre los dispositivos terminales, dichos nodos solo requerirán llegar hasta la capa de red como se muestra a continuación:



**Ilustración 3: Comunicación con nodos intermedios**

Para el proyecto de la memoria es necesario modificar las características de la capa de transporte, más precisamente lo relacionado a *TCP*. Esto porque la capa de transporte es donde ocurre un cuello de botella de transferencia de datos y además porque no requiere cambiar los componentes de la red (como sería el caso de modificar capas inferiores) ni tampoco que sea dependiente y exclusivo del software a utilizar (en caso de modificar las capas superiores). Cabe tener en cuenta que a pesar de que el modelo *TCP/IP* tiene su propio modelo de capas, para el desarrollo de la presente memoria se seguirá haciendo referencia al modelo OSI pues en este caso particular resulta prácticamente indiferente dado que la capa de transporte es compartida.

## 2.3 Capa de transporte

La capa de transporte tiene como propósito entregar una transferencia confiable de datos entre estaciones terminales, despreocupando de dichas necesidades para un buen funcionamiento de la comunicación a las capas superiores.

Los servicios prestados por esta capa varían de acuerdo al protocolo específico usado y entre ellos pueden estar:

- Comunicación orientada a la conexión.
- Confiabilidad de la conexión.
- Control de tráfico.
- Control de evasión de congestión.

## 2.4 Protocolo IP

Una red de telecomunicaciones corresponde a un conjunto de terminales, links (enlaces) y nodos que permitan la comunicación entre distintos equipos terminales. Para el interés de la memoria es importante destacar que solo interesa el tipo de conmutación por paquetes, de este modo además la comunicación entre terminales será descentralizada.

Las redes IP son redes de conmutación de paquetes. El protocolo IP está diseñado para proveer transmisión de bloques de datos binarios llamados datagramas desde un emisor a un receptor en donde ambos son *host* identificados por una dirección de largo fijo.

Este protocolo por si solo está limitado en cuanto a entregar las capacidades necesarias para enviar los datagramas de la fuente al destino sin entregar garantías de alcanzar al cliente final, es decir, al enrutamiento de los datos. No posee mecanismos para aumentar la fiabilidad de datos entre los extremos, control de flujo, secuenciamiento u otros servicios que se encuentran normalmente en otros protocolos *host to host*. Dado a esto su funcionalidad corresponde a la capa de red del modelo OSI. La inteligencia de la red se encuentra en los dispositivos terminales y no en los componentes internos de la red.

En este protocolo hay que diferenciar los nombres de las direcciones y de las rutas. Un nombre indica lo que se desea buscar, la dirección IP se refiere a donde está, y la ruta indica el camino para llegar. IP maneja solo direcciones y corresponde a los protocolos de las capas superiores hacer la correspondencia entre nombres con direcciones.

## 2.4.1 Comparación de conmutación de paquetes de Redes IP con redes de conmutación de circuitos

Las redes de conmutación de circuitos corresponden a aquellas en donde existe un canal dedicado y exclusivo para un par de terminales formándose un “camino físico” entre ellas. Este tipo de conmutación es clásica de la telefonía tradicional y tienes notables comparaciones con la conmutación de paquetes utilizada en IP:

Tabla 1: Comparación entre conmutación de circuitos versus conmutación de paquetes

Conmutación de circuitos	Conmutación de paquetes
Equipos terminales sin inteligencia y ésta recae en la red	La inteligencia se encuentra en los equipos terminales y no en la red.
Camino único durante la comunicación	Los paquetes pueden tomar caminos distintos
Calidad de servicio garantizada	Calidad de servicio no garantizada, se trabaja en lo que se conoce como <i>best effort</i>

Esta última comparación es de interés para proyecto pues dado que se trabaja en *best effort* igual se requiere algún método para mantener el servicio estable.

## 2.5 TCP

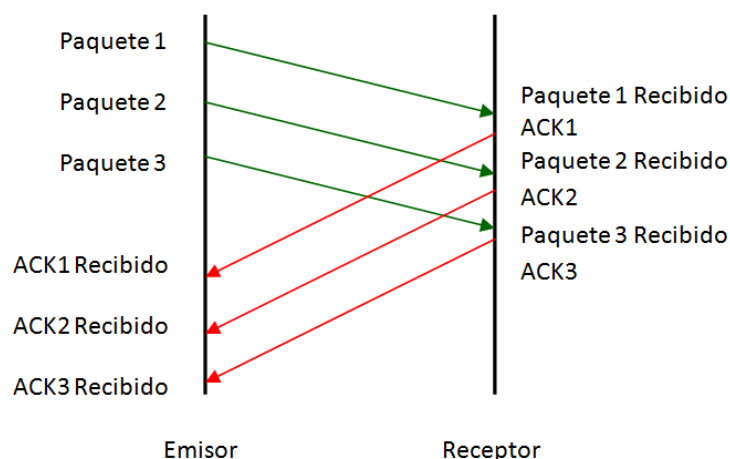
*TCP* corresponde al protocolo del modelo IP de la capa de transporte orientado a la conexión. Este protocolo se encarga de que la conexión sea fiable, segura, libre de errores y justa.

### 2.5.1 Funcionamiento de TCP en la transferencia de datos

Una conexión *TCP* tiene tres pasos: establecimiento de la conexión, transferencia de datos y término de la conexión. La más importante para efectos de la memoria corresponde a la etapa de transferencia de datos pues allí es donde estarán funcionando el protocolo de control de congestión.

En la transferencia de datos ocurren una serie de procesos que *TCP* usa para entregar seguridad y confiabilidad a la conexión. Entre ellos los más importantes son: El uso de números de secuencia (tanto de emisor como de receptor) y el escalamiento de la ventana de congestión.

Durante la transferencia *TCP* el emisor envía un paquete al terminal receptor. Este paquete está enumerado por su número de secuencia (*SEQ*) y si es recibido correctamente por parte del receptor, éste envía otro número de secuencia, llamado número de asentamiento o *ACK* (cuando se trata de receptor a emisor) que se usa como verificador de la llegada del paquete. Así mismo, los números de secuencia sirven también para ordenar los paquetes recibidos en caso de que lleguen varios y el orden de arribo no corresponda al orden de envío.



**Ilustración 4: Esquema de transferencia TCP**

Los *ACKs* además se usan por parte de la entidad emisora como medidores del estado de la red. Esto puede ser usado mediante el tiempo de demora de los *ACK* o la ausencia de éstos. En *TCP* el emisor envía datos a lo que el receptor es capaz de recibirlos.

En *TCP* la cantidad de paquetes enviados que podría estar circulando por red dado que aún no se reciben su correspondiente *ACK* se denomina ventana de congestión (abreviado en inglés como *cwnd*), esta ventana es variable y se determina su valor de acuerdo al estado de la red mediante diversos métodos, entre ellos los algoritmos para evitar congestión.

## 2.6 Algoritmos para evitar congestión

Actualmente las conexiones de Internet son bastante comunes y siguen expandiéndose a niveles tanto en tamaño, diversidad y su integración con otras redes. Por lo mismo el entendimiento y control de este recurso es crucial. La escala de la Internet a nivel tamaño es muy extensa y además su complejidad no es menor. También hay que considerar que la restricción de que se trata de una red descentralizada hace imposible realizar un control óptimo puesto que los servidores y enlaces solo tienen accesos a su información local.

Para realizar un control para evitar congestión primero se requiere decidir cómo medir este concepto para ser modelado [17]. Una manera corresponde a que asignándole a cada enlace un valor escalar llamado “precio” que mide su congestión. Este valor depende de otros parámetros como pueden ser la probabilidad de perder paquetes en dicho enlace, el retardo de encolamiento en el *router* del enlace, etc.

Un protocolo de control de congestión consiste entonces en un algoritmo que actúa durante una parte de la transferencia de datos cuya función corresponde a evitar, tal como dice su nombre, un colapso de la red por congestión y en caso de que la congestión se encuentre presente encargarse de dejar la red en su estado normal. En términos generales dependiendo del tipo de red estos métodos de evasión de congestión pueden modificar la terminal receptora, terminal emisora de datos (servidor) o las características de la red (*routers* por ejemplo). Específicamente en *TCP* por la arquitectura de Internet, estos métodos modifican únicamente las características del servidor y trabajan en la capa cuatro del modelo OSI. Otro punto a considerar es que no hay un método que sirva para todos los casos, estos están orientados a algún tipo específico de red con características muy claras.

En *TCP* el estudio que más interesa para el presente trabajo corresponde a caracterizar las condiciones de equilibrio del método para evitar congestión desde el punto de vista de “justicia” (*fairness*), uso eficiente de recursos y dependencia de los parámetros de la red, así como también de la velocidad de convergencia.

*Fairness* se refiere a que el protocolo reparte de manera justa los recursos de la red [2], notar que dado que estos métodos algoritmos trabajan en el servidor, otra manera de explicar lo anterior se refiere a que el servidor reparta de manera justa los recursos de la red a sus clientes. De no cumplirse el *fairness* algunos clientes consumirán más recursos y otros, dado lo anterior, creerán por parte del servidor que la red esta congestionada cuando no es el caso, recibiendo una cantidad menor de recursos de los que debería. Una definición más técnica vendría siendo: el ancho de banda asignado a un flujo específico no puede aumentar sin disminuir el ancho de banda asignado para un flujo con igual o menor tasa de transferencia.

Claramente los objetivos principales del control de congestión se realizan imperfectamente en lo que se conoce como “*best effort*” internet. Sin embargo dado el *fairness* como parámetro de diseño de los mecanismos de control de congestión permite que el rendimiento del *throughput* sea bastante robusto y explica porque el “*best effort*” entrega tan buen servicio [2].

En cuanto a la dependencia de los parámetros de la red se refiere principalmente a de qué manera se caracterizara el estado de la red. Como se mostrara más adelante, dependerá del algoritmo específico a utilizar y estos parámetros son la tasa de pérdida de paquetes o el retardo de la red o desviaciones estándar de los mismos.

De acuerdo al algoritmo de control de congestión a utilizar pueden percibirse cambios drásticos en cuanto a la tasa de transferencia [3], sin embargo hay que tener en cuenta que el *throughput* finalmente depende de una diversidad de factores y de acuerdo al contexto y

naturaleza física de la transmisión puede darse el caso que el método de congestión a utilizar no sea relevante [12].

## 2.6.1 Algunos Algoritmos de control de congestión utilizados:

### 2.6.1.1 RENO

Corresponde a la implementación predominante de *TCP* [17]. Consta de tres acciones que modifican la ventana de congestión a diferentes maneras: El primer caso se llama *slow-start* y ocurre cuando la transmisión está iniciando o un *ACK* del receptor no llega en el tiempo límite y ocurre un *time-out*, en este caso la ventana de congestión se inicializa con un valor muy pequeño y para cada *RTT* el valor se duplica hasta llegar a un valor umbral *ssthresh* en donde *slow-start* termina; El segundo caso es controlado por un parámetro llamado *Additive Increase* (AI) que aumenta la ventana de congestión linealmente en un número fijo de paquetes por cada *RTT* (cada vez que se recibe un *ACK* del receptor) de esta manera se obtiene un aumento de la ventana suave; En el último caso el parámetro de interés se denomina *Multiplicative Decrease* (MD) y consiste en reducir la ventana de congestión cuando se presenta el caso de pérdida de paquetes. En Reno  $AI=1$  y  $MD=0.5$ , es decir la ventana aumenta de un paquete a la vez por *RTT* fuera de la fase *slow-start* y cuando se detecta congestión la ventana se reduce a la mitad.

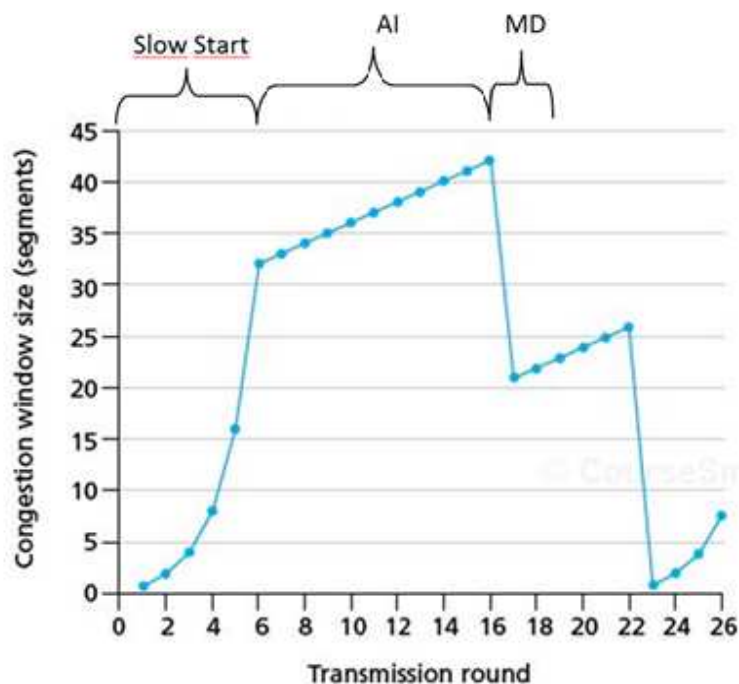


Ilustración 5: Gráfica de ejemplo del comportamiento de TCP Reno

En lo referente a la etapa *slow-start*, el umbral *ssthresh* se inicializa con un valor dado para una nueva transmisión, pero para una transmisión en transcurso en donde se llega a *slow-start* por *time-out* el umbral corresponde al último valor en donde ocurrió congestión y la

ventana disminuyo por el MD, sin embargo existen propuestas para mejorar este procedimiento pues el *slow-start* puede inducir a ráfagas grandes de paquetes y comportamientos no deseables en el flujo de datos [10].

### 2.6.1.2 VEGAS

TCP VEGAS [17] corresponde a un método de evasión y control de congestión cuyo parámetro de control corresponde al *RTT* y en menor medida a la pérdida sufrida. También tiene tres mecánicas principales: la primera se refiere a que posee un mecanismo de retransmisión al primer ACK duplicado y no al segundo como el caso de RENO, resultando en una detección de pérdida más oportuna; la segunda mecánica se refiere a una forma más prudente de aumentar la ventana de congestión durante el *slow-start* resultando en menos pérdidas; y el tercer mecanismo corresponde al método de evasión de congestión que trata de corregir el comportamiento oscilatorio de RENO, el funcionamiento está basado en periódicas mediciones del *RTT* y de acuerdo a la constante de proporcionalidad entre éste y el retardo de encolamiento se altera la ventana de congestión.

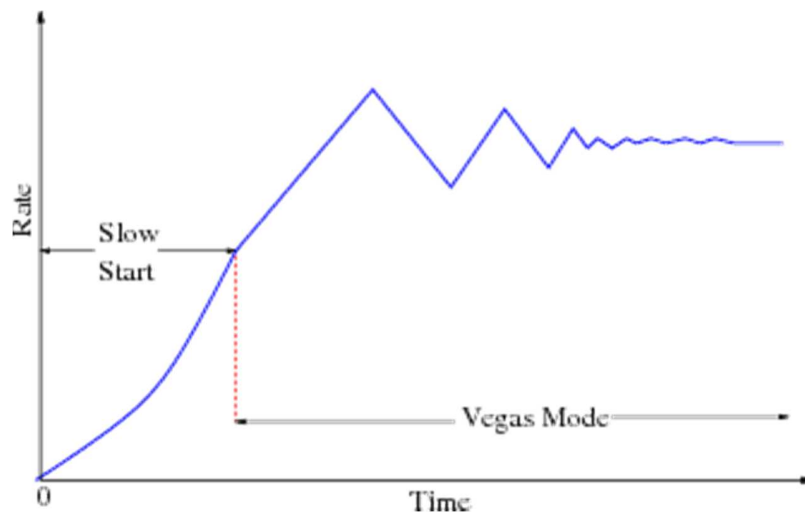


Ilustración 6: Gráfica de ejemplo del comportamiento de TCP Vegas

### 2.6.1.3 BIC y CUBIC

BIC y CUBIC [8] corresponden a implementaciones de control de congestión desarrollados por *Networking Research Lab* cuya función consiste en suplir el bajo rendimiento de *TCP* bajo ciertas condiciones. El objetivo de estos protocolos es que puedan escalar de acuerdo a rendimiento hasta *throughputs* elevados por redes de alta velocidad de transferencia manteniendo el *fairness*, estabilidad y *TCP-friendliness*.

En BIC la ventana de congestión varía de la siguiente manera: cuando ocurre un evento de congestión la ventana se reduce de acuerdo a un valor multiplicativo. El tamaño de la ventana justo antes de su disminución de tamaño se establece como máximo y el tamaño de la ventana justo después de la reducción se ajusta al valor mínimo. Entonces, BIC realiza una

búsqueda binaria utilizando estos dos parámetros (máximo y mínimo) cayendo al "punto medio" entre el máximo ( $W_{max}$ ) y el mínimo ( $W_{min}$ ). Puesto que las pérdidas de paquetes se han producido en  $W_{max}$  el tamaño de la ventana que actualmente la red puede manejar sin pérdida debe estar en algún lugar entre estos dos números.

Sin embargo, saltar al punto medio podría ser un excesivo aumento de la ventana en el transcurso de un *RTT*, por lo que si la distancia entre el valor medio y la ventana mínima es mayor que una constante fija predefinida, llamada  $S_{max}$ , BIC incrementa el tamaño de la ventana actual por  $S_{max}$ , fase que se denomina *additive increase*. Si BIC no recibe las pérdidas en el tamaño de la ventana actualizada, dicho valor se convierte en el nuevo  $W_{min}$ . Por el contrario, si llegara a ocurrir una pérdida de paquetes, el tamaño de la ventana se convierte en el nuevo  $W_{max}$ . Este proceso continúa hasta que el crecimiento de la ventana es menor que una constante pequeña predefinida llamada  $S_{min}$  en cuyo punto, la ventana se establece como  $W_{max}$ . Esto produce un crecimiento logarítmico de la ventana y se denomina *binary search*.

Si la ventana aumenta más allá del máximo, el tamaño de la ventana de equilibrio debe ser mayor que  $W_{max}$  de ese entonces y un nuevo máximo debe ser encontrado. BIC entra en una nueva fase denominada *max probing*. La función de crecimiento durante esta fase es inversa de los de *binary search* y *additive increase*. Crece exponencialmente y luego linealmente. *Max probing* utiliza una función de crecimiento para la ventana exactamente simétrica a los utilizados en aumento binario (*binary search* y *additive increase*) pero en un orden inverso.

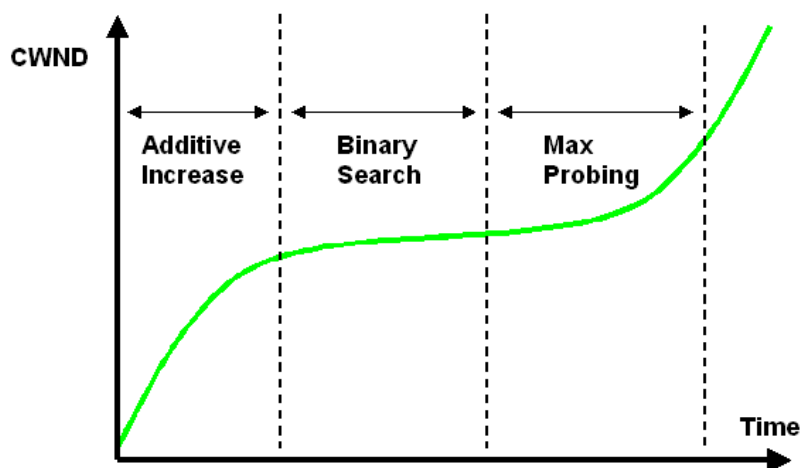


Ilustración 7: Gráfica de ejemplo del comportamiento de TCP BIC

BIC tiene un problema referente a que la función de crecimiento de la ventana de congestión puede resultar bastante agresiva para *TCP* bajo ciertas circunstancias como *RTT* muy pequeños o redes de baja tasa de transmisión. De allí *Networking Research Lab* desarrolló una nueva versión de BIC con una nueva función de crecimiento para la ventana: una función cúbica. Esta función crece más lento cerca de  $W_{max}$  que *binary search* y además en este protocolo de control no existe la fase *additive increase*.



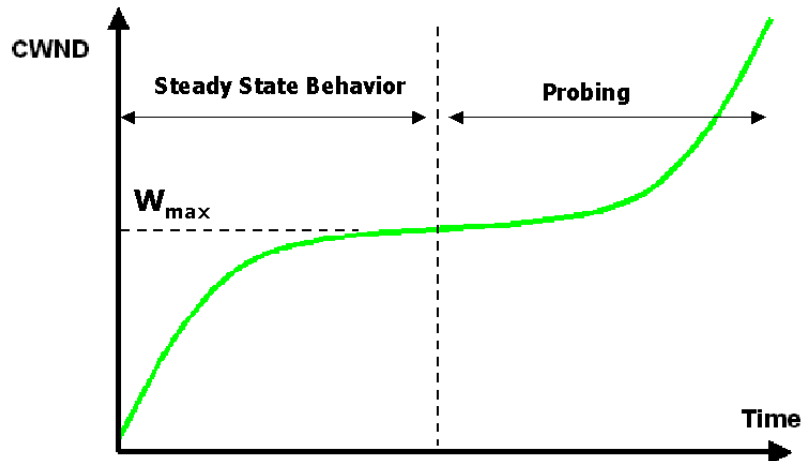


Ilustración 8: Gráfica de ejemplo del comportamiento de TCP Cubic

## 2.7 Estimación teórica del *throughput* Reno

El *throughput* de TCP usando el algoritmo de evasión de congestión clásico (Reno) corresponde a [19]:

$$\text{Throughput} = \frac{1}{RTT \sqrt{\frac{2bp}{3}} + T_0 \min\left(1, 3 \sqrt{\frac{3bp}{8}}\right) p(1 + 32p^2)}$$

Donde  $RTT$  corresponde al retardo en milisegundos,  $p$  corresponde a la pérdida en por unidad,  $b$  corresponde a la cantidad de  $RTT$ s para incrementar la ventana (también se puede interpretar como la cantidad de  $ACK$ s a esperar para incrementar la ventana) y  $T_0$  corresponde al tiempo de *time-out*. Esta fórmula puede ser dividida en dos partes de acuerdo al tipo de pérdida:

Pérdida por *triple-duplicated ACK*:

$$\text{Throughput}_{TD} = \frac{1}{RTT \sqrt{\frac{2bp}{3}}}$$

Pérdida por *time-out*:

$$\text{Throughput}_{TO} = \frac{1}{T_0 \min\left(1, 3 \sqrt{\frac{3bp}{8}}\right) p(1 + 32p^2)}$$

En proyecto contempla solo el primer tipo de pérdida porque en el entorno de pruebas a utilizar es muy difícil y poco probable que ocurran pérdidas por *time-out*. Además también se considerara  $b=1$ , es decir que por cada *RTT* la ventana crece en un valor. Entonces la fórmula de *throughput* final que será utilizada para comparar con los resultados prácticos es:

$$Throughput = \frac{1}{RTT \sqrt{\frac{2p}{3}}}$$

## 2.8 Modelo AIMD

El modelo AIMD [16] corresponde a un caso general de Reno en donde a diferencia de éste los parámetros AI (*additive increase*) y MD (*multiplicative decrease*) no son fijos en 1 y 0.5. Hay una restricción que dichos valores de AI y MD deben cumplir entre ellos para que la transmisión sea *TCP-friendly*. En este sentido el modelo AIMD es mucho más general pues no corresponde a un control de evasión de congestión Reno con valores distintos, va más allá: los valores de AI y MD puede ser dinámicos y acomodarse al estado del canal, o pueden ser definidos a priori sabiendo el tipo de topologías de la red, lo importante es que la relación entre ellos se cumpla en todo instante pues la inclusión del *TCP-friendly* es necesaria.

Hay dos formas de relacionar AI y MD, una considerando pérdidas debido a *triple-duplicated ACK* y otra a pérdidas por *time-out*.

Por *time-out*:

$$AI = \frac{4(1 - MD^2)}{3}$$

Por *triple-duplicated ACK*:

$$AI = 3 \frac{1 - MD}{1 + MD}$$

Considerando las siguientes restricciones:

$$0 \leq AI$$

$$0 \leq MD \leq 1$$

Por contraproducente que parezca se utiliza mucho en la literatura la formulación obtenida por *time-out*, para ser más acordes al proyecto se utilizara la formula obtenida por *triple-duplicated ACK* que además es algo más restrictiva; usar la otra formulación puede entregar resultados *TCP-friendly* pero al momento de verificación practica no lo son.

## 2.9 Controlador Proporcional Integral [18]

Definiciones básicas de control:

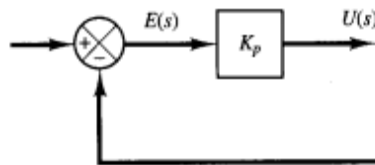
- Variable controlada: Corresponde a la cantidad o condición que se mide y controla. Comúnmente la variable controlada es la salida del sistema.
- Variable manipulada: Corresponde a la cantidad o condición que el controlador modifica para afectar el valor de la variable controlada.
- Controlar: Significa medir el valor de la variable controlada del sistema y actuar sobre la variable manipulada al sistema para corregir o limitar una desviación del valor medido a partir de un valor deseado.

Un PI corresponde a un controlador retroalimentado que ajusta la variable manipulada de acuerdo a la desviación del error, definido como la diferencia entre la referencia o valor deseado de la variable controlada con el valor actual. Para este tipo de controlador desviación corresponde al error instantáneo (parte proporcional) como al error acumulado en el tiempo (parte integral).

En la acción de control proporcional la relación entre la salida del controlador  $u(t)$  y la señal de error  $e(t)$  es:

$$u(t) = K_p e(t)$$

Con  $K_p$  la ganancia proporcional. Como se muestra el controlador proporcional no es más que un amplificador con una ganancia ajustable.

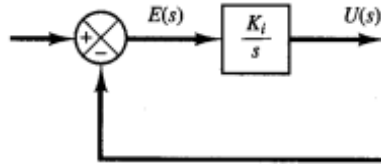


**Ilustración 9: Diagrama de bloques de un controlador proporcional.**

En la acción de control integral, el valor de la salida del controlador  $u(t)$  se cambia a una razón proporcional a la señal de error  $e(t)$ :

$$u(t) = K_i \int_0^t e(t) dt$$

Con  $K_i$  la constante integral del controlador.

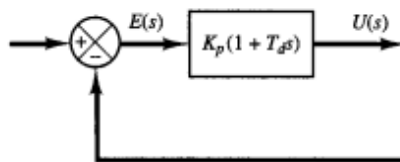


**Ilustración 10: Diagrama de bloques de un controlador integral**

La función del integrador es disminuir y eliminar el error de estado estacionario.

El PI consiste entonces de un controlador con tanto la parte proporcional como la integral:

$$u(t) = K_p e(t) + K_i \int_0^t e(t) dt$$



**Ilustración 11: Diagrama de bloques de un controlador PI**

## 2.10 High bandwidth delay product Networks

*Bandwidth-Delay Product* se refiere al producto de la capacidad del enlace (en bits por segundo) por el retardo entre las terminales interactuando (en segundos) y como interpretación corresponde a la cantidad de datos circulantes en la red, en otras palabras datos que están siendo transmitidos en el canal y aun no son recibidos. Por lo tanto, bajo esta definición las redes *high bandwidth delay product* corresponden a redes cuya capacidad de transmisión de datos es muy elevada y a la vez el *RTT* de dichos enlaces también poseen valores altos.

En este tipo de redes es donde el protocolo *TCP* tiene un pobre desempeño por su lento crecimiento de la ventana de congestión [21][13] (un paquete por *RTT* lo que es equivalente a  $AI=1$ ) y por el rápido decremento de ésta (*multiplicative decrease* =0.5) [7]. Por lo mismo, mejorar el rendimiento y eficiencia de recursos para la transferencia de datos de este tipo de redes es un área de mucho estudio y en la literatura hay muchas alternativas propuestas como soluciones: *TCP Parallelisation* [7] en donde se utilizan un set de conexiones *TCP* paralelas para transmitir datos, modificar protocolos de control ya conocidos y adaptarlos a las necesidades de redes *high bandwidth delay product*[22][14][21], diseñar protocolos de control específicos para este tipo de redes[20][5][9][11].

## 2.11 Estimación de *throughput* como herramienta

Estimar la tasa de transferencia de manera precisa corresponde a una herramienta bastante poderosa al momento de optimizar los recursos de la red. Partiendo por el hecho de que en el proyecto la manera de estimar el *throughput* corresponde a un protocolo de control de congestión en donde utilizando los parámetros dinámicos de la red (*RTT*, porcentaje de pérdida de paquetes) la ventana de congestión convergerá a la tasa de transferencia. Esto quiere decir que es posible usar esta utilidad de dos formas distintas: como un protocolo de evasión de congestión independiente y que cumple con todos los requisitos (*fairness* y *TCP friendly*); o como un instrumento de ayuda para otros métodos de control de congestión.

En redes *high bandwidth delay product* conocer tasa de transferencia a priori permite un uso más eficiente de la red dado que es posible definir y encontrar la cantidad de paquetes a transmitir para llegar a la capacidad del canal y así obtener un rendimiento superior a *TCP*. Por lo tanto el proyecto de memoria consiste también en una solución alternativa a las propuestas en la literatura para los problemas presentes en la transmisión en redes *Bandwidth-Delay Product*.

# CAPÍTULO 3: IMPLEMENTACIÓN

## 3.1 Introducción

En este capítulo se presenta la metodología de implementación utilizada en el proyecto. Como se verá consta de dos grandes partes: la etapa de simulación correspondiente a la programación del algoritmo de evasión de congestión en el software OPNET de manera que entregara una primera impresión del funcionamiento del código respecto a lo esperado teóricamente; luego está la etapa de implementación propiamente tal en entorno Linux en donde tiene que quedar funcionando sin inconvenientes para la realización de pruebas de laboratorio controladas.

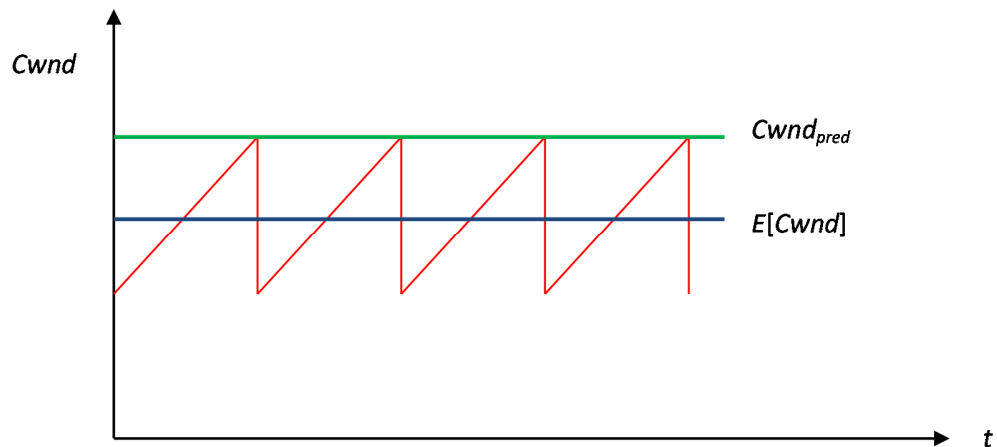
Los códigos tanto de OPNET como del módulo de Linux están mostrados en la sección final de Anexos.

Hay dos clasificaciones distinguibles dentro del funcionamiento de los algoritmos a implementar para controlar la convergencia a la ventana de congestión.

- El primer método consiste en considerar un valor de  $\alpha$  porcentual a la diferencia entre la ventana de congestión y la ventana predicha y luego el  $\beta$  correspondiente se obtiene aplicando la relación existente entre  $\alpha$  y  $\beta$  para que el protocolo sea *TCP-friendly*. El módulo de este método queda referido como *tcp\_conn\_alt*.
- El segundo método se trata de un controlador PI con variable manipulada  $\alpha$  de modo de controlar la ventana de congestión. También en este caso se obtendrá  $\beta$  para que el protocolo sea *TCP-friendly*. El módulo de este método queda referido como *tcp\_conn\_pi*.

## 3.2 Predicción de la ventana de congestión

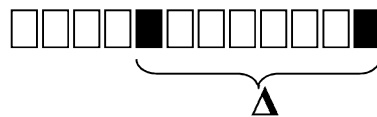
La ventana de congestión predicha corresponde a la capacidad del canal. Teóricamente corresponde a  $\frac{4}{3}$  del valor esperado de la ventana de congestión del algoritmo Reno. Esto se deduce considerando pérdida a tasa constante de la siguiente manera:



**Ilustración 12: Ventana de congestión de Reno, Ventana de congestión predicha y Valor esperado de la ventana de congestión**

Para el funcionamiento del algoritmo es necesario conocer la referencia  $cwnd_{pred}$  a la cual deba converger la ventana de congestión. Para ello se plantea un método que va actualizando la predicción a medida que transcurre el tiempo y van ocurriendo pérdidas.

El método propuesto para definir la ventana de congestión predicha se basa en una estimación simple del nivel de congestión en el canal. Definiendo  $\Delta$  como la diferencia entre el instante actual y el momento de la última pérdida sufrida (corresponde al recíproco de la pérdida instantánea y sirve como estimador del nivel de pérdida en el canal en dicho momento) y tomando la ecuación de *throughput*:



**Ilustración 13:  $\Delta$  como distancia entre dos pérdidas.**

$$T[pkts] = \frac{1}{RTT \sqrt{\frac{2p}{3}}}$$

$$E[cwnd] = \frac{1}{\sqrt{\frac{2p}{3}}}$$

$\Delta$  corresponde al recíproco de la pérdida instantánea, por lo tanto:  $\Delta = \frac{1}{p}$

$$E[cwnd] = \sqrt{\frac{3\Delta}{2}}$$

$$cwnd_{pred} = \frac{4}{3}E[cwnd] = \frac{4}{3}\sqrt{\frac{3\Delta}{2}} = \sqrt{\frac{8\Delta}{3}}$$

$$\therefore cwnd_{pred \text{ instantanea}} = \sqrt{\frac{8\Delta}{3}}$$

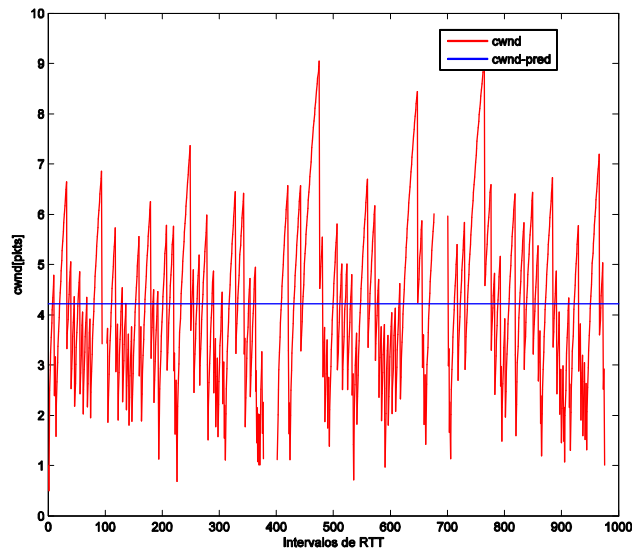
Este valor es instantáneo y por lo tanto la predicción de la ventana corresponde a un promedio ponderado:

$$\begin{aligned} cwnd_{pred(n)} &= (1 - w)cwnd_{pred(n-1)} + w \cdot cwnd_{pred \text{ instant}(n)} \\ &= (1 - w)cwnd_{pred(n-1)} + w \cdot \sqrt{\frac{8\Delta}{3}} \end{aligned}$$

Con  $w$  el peso asociado a la última medición. Con  $w=1/n$ ,  $cwnd_{pred(n)}$  corresponde al promedio de todas las predicciones obtenidas a medida que el modulo va funcionando:

$$cwnd_{pred(n)} = \sum_{i=1}^n \frac{cwnd_{pred \text{ instant}(i)}}{n} = \frac{(n-1)cwnd_{pred(n-1)} + cwnd_{pred \text{ instant}}}{n}$$

Utilizando esta predicción el resultado debería parecerse a:



**Ilustración 14: Rojo: Ventana de congestión de Reno. Azul: Ventana predicha**



Luego, conociendo la predicción es necesario modificar de manera inteligente la ventana para converger. Se modificaran los parámetros  $\alpha$  y  $\beta$  de acuerdo a los métodos propuestos siempre manteniendo *TCP-friendly* para llevar la ventana de congestión al valor deseado.

### 3.3 Convergencia a la capacidad del canal

Una vez conocida la capacidad de transferencia del canal mediante el método propuesto es necesario llevar el tamaño de la ventana de congestión a dicho valor. En principio no debería ser necesario algún método adicional más que forzar la ventana a la capacidad del canal salvo la estricta razón que el protocolo debe ser justo, y simplemente forzar la ventana a la capacidad no asegura el *fairness*. Una manera de asegurarse la justa repartición de recursos del canal es utilizando la relación *fairness* de los modelos de evasión de congestión AIMD que relaciona  $\alpha$  y  $\beta$ . Luego la manera de converger a la tasa de transferencia predicha consiste en modificar eficientemente  $\alpha$  de acuerdo a la distancia entre la ventana de congestión y la predicha, y para instantes de pérdida reducir la ventana de acuerdo al  $\beta$  *fair* correspondiente entregado por la relación.

#### 3.3.1 Método de diferencia porcentual

Este método para converger a la ventana considera  $\alpha$  como:

$$\alpha = \max \left\{ \frac{(cwnd_{pred} - cwnd)}{cwnd_{pred}}, 0 \right\}$$

Dado que  $\alpha$  no puede ser negativo se limita su valor mínimo a 0, esto ocurre cuando la ventana de congestión resulta más grande que la ventana de congestión predicha.

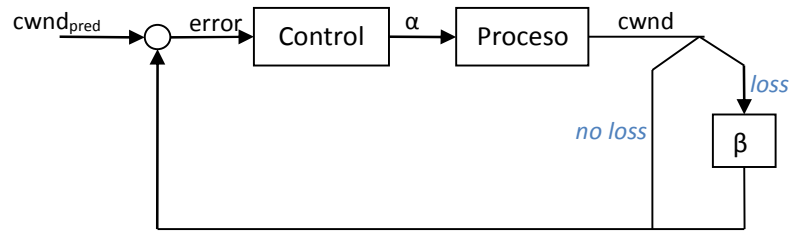
Por otro lado,  $\beta$  se considera como:

$$\beta = \min \left\{ \beta_F, \frac{cwnd_{pred}}{cwnd} \right\}$$

Donde  $\beta_F$  corresponde al  $\beta$  obtenido de la relación de *fairness*. En este caso particular es necesario tener precaución y se decide entre el mínimo de dicho valor y  $\frac{cwnd_{pred}}{cwnd}$  por la razón que bajo circunstancias donde  $cwnd > cwnd_{pred}$  el valor de  $\alpha$  es 0, usando la relación de *fairness* para  $\alpha = 0$  se obtiene  $\beta = 1$ , es decir la ventana de congestión no decrece en instantes de pérdida lo cual incurre a problemas pues no habría convergencia. Bajo la formulación de  $\beta$  propuesta, cuando exista pérdida sobre el umbral de ventana de congestión predicha el valor de  $\beta$  será tal que deje en un paso la ventana al valor deseado. Esta formulación no trae inconvenientes con el *fairness* puesto que las opciones a elegir son un valor *fair* ( $\beta_F$ ) o un valor más *fair* incluso  $\left(\frac{cwnd_{pred}}{cwnd}\right)$ .

### 3.3.2 Método usando un controlador PI

El método anterior puede ser visto también como un controlador proporcional con variable controlada la ventana de congestión, variable manipulada  $\alpha$  y “constante” de multiplicación del controlador  $\frac{1}{cwnd_{pred}}$  o 0 dependiendo si se está sobre o bajo la predicción, que en si no es una constante pues cambia para cada nuevo cálculo de  $cwnd_{pred}$ . El sistema es retroalimentado y se representa como sigue:



**Ilustración 15: Esquema de control para el sistema**

Desde el punto de vista de control de sistemas un control proporcional puede tener error de estado estacionario no nulo porque mientras más se acerca a la referencia menos varía la variable controlada. Por lo mismo también se utiliza un controlador PI para lograr la convergencia a la ventana predicha que corresponde a la referencia del sistema de control. Hay que hacer notar que utilizando un controlador PI en este caso particular no asegura error de estado estacionario nulo por una restricción netamente del sistema operativo que será expresada en su momento.

El proceso queda representado como sigue:

$$cwnd(n + 1) = cwnd(n) + \frac{\alpha}{cwnd(n)}$$

Sin embargo considerando la integral del error es posible llegar a la referencia. Además a pesar de tener un error estacionario no nulo, si la media de la ventana de congestión usando este sistema de control corresponde a la referencia (ventana predicha) y las oscilaciones presentes están dentro de ciertos márgenes se considerará un buen resultado.

Este controlador PI, es de la forma  $G_k = K_p e(t) + K_i \int_0^t e(\tau) d\tau$  para sistemas de variable temporal continua y de la forma  $G_k = K_p e[n] + K_i \sum e[n]$  para sistemas de tiempo discreto como el presente caso. Por tratarse de un sistema no lineal las constantes proporcional e integral no se pueden encontrar analíticamente, por lo mismo se realizan pruebas con el módulo de Linux operativo para dos variables distintas:  $K_p$  y  $R = \frac{K_i}{K_p}$ . En la sección de análisis

de resultados se muestran comportamientos tales como media y varianza de la ventana de congestión para distintos valores de  $K_p$  y  $R$ .

En este método para converger a la ventana se considera

$$\alpha = \max \left\{ K_p e(t) + K_I \sum e[n], 0 \right\}$$

Con

$$e[n] = (cwnd_{pred}[n] - cwnd[n])$$

Al igual que el método de diferencia porcentual hay que considerar el máximo entre  $K_p e[n] + K_I \sum e[n]$  y 0 dado que  $\alpha$  no puede ser negativo.

$\beta$  se considera igual que utilizando el método anterior como:

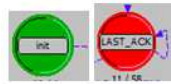
$$\beta = \min \left\{ \beta_F, \frac{cwnd_{pred}}{cwnd} \right\}$$

Además, se agregó una característica especial en donde si ocurre un estado de *slow-start*, el error de integración se reinicia a valor 0. La razón de esto radica en que la ventana de congestión puede elevarse demasiado cuando ocurre un *slow-start* disparando el error y también la integral de éste.

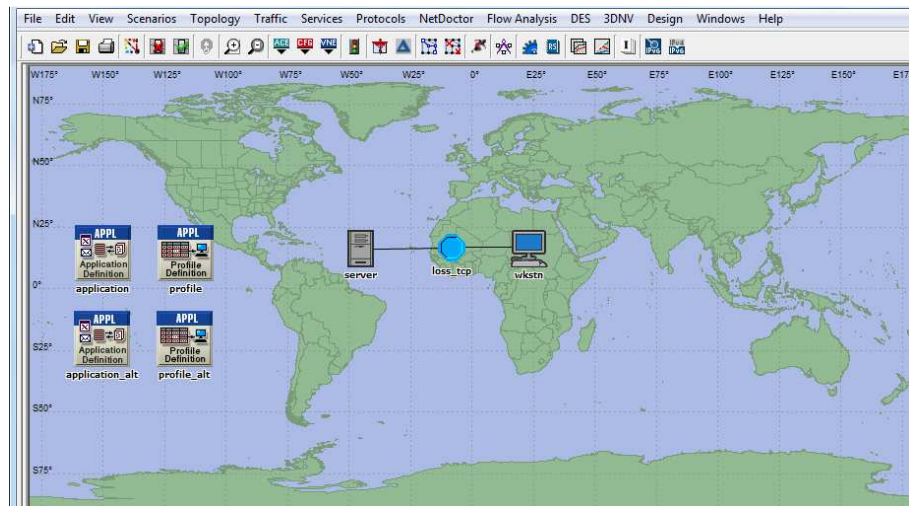
### 3.4 Simulación OPNET

OPNET corresponde a un software extremadamente poderoso en donde se puede trabajar con protocolos de transporte y realizar un sinnúmero de simulaciones. Para este caso específico se realizó un escenario en donde existe un servidor, un cliente y un nodo central encargado de generar pérdida de paquetes. Se utiliza como base de trabajo el protocolo Reno en donde se reescribe parte de este hasta obtener los algoritmos deseados que serán simulados. Una vez que la simulación de resultados coherentes se procede a la implementación final en Linux.

La forma de programar en OPNET es nodal: se presentan nodos de los cuales algunos tienen condiciones, los nodos no forzados (de color rojo) tienen dos partes de código, si se llega al nodo se corre la primera parte de éste, si se cumple la condición del nodo éste además corre el código de la segunda sección. Los nodos forzados (de color verde) no tienen condiciones y si se llega a estos su código se ejecuta. Los nodos pueden estar conectados para así tener asignado el orden.

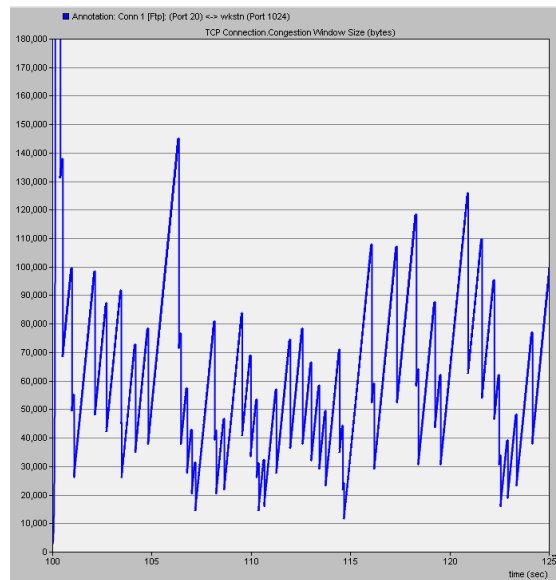


**Ilustración 16: Nodo forzado de color verde y nodo no forzado de color rojo**



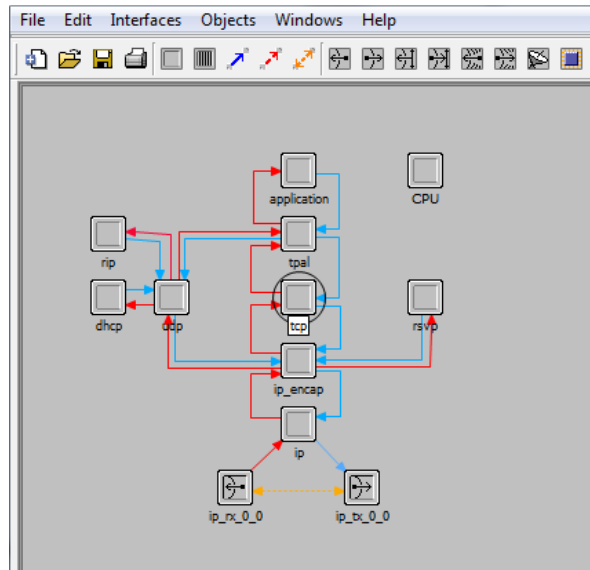
**Ilustración 17: Escenario OPNET de simulación de Reno**

Primero se realiza una simulación para corroborar el funcionamiento del software usando el protocolo Reno con una probabilidad de pérdida 0.001 con distribución uniforme y un RTT de 10[ms].



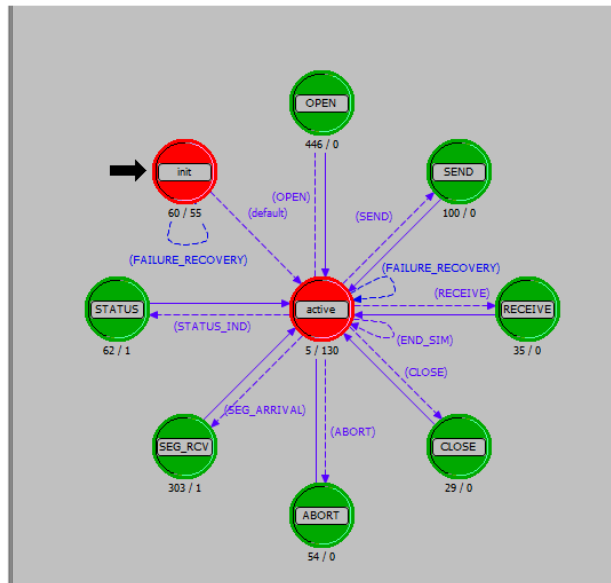
**Ilustración 18: Simulación usando algoritmo Reno, pérdida de 0.1% y RTT de 10[ms]**

Para modificar el algoritmo de evasión de congestión primero hay que abrir el modelo nodal del servidor:



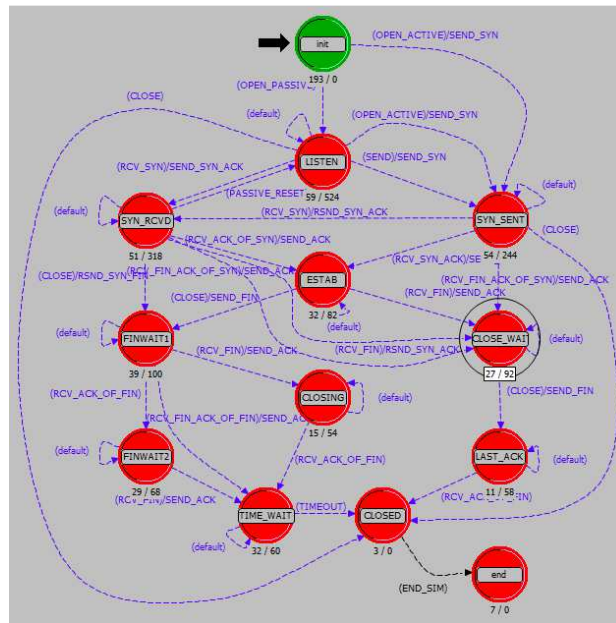
**Ilustración 19: Modelo nodal del servidor**

Luego abrir el CPU de nombre *TCP*:



**Ilustración 20: Modelo nodal de TCP**

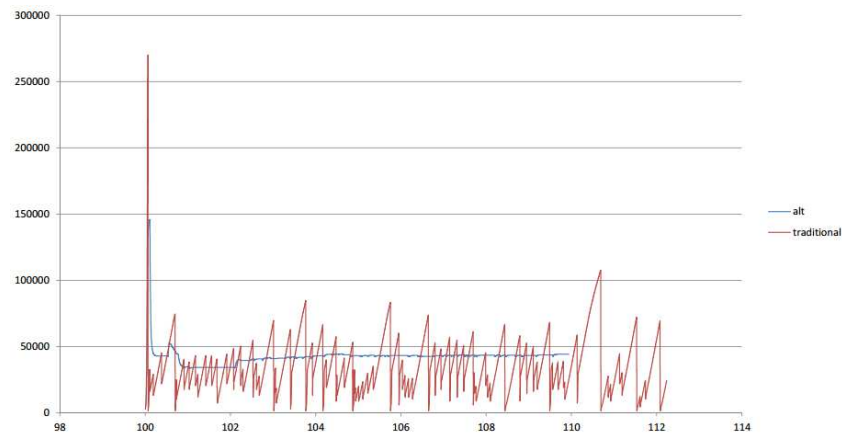
Se presenta *TCP* desde su visión global. Se necesita entrar más aun en detalle, *TCP* es muy amplio y para el proyecto solo es necesario el segmento de protocolos de transporte, para lo cual desde la misma ventana se va a *File->Open Child Process Model* llegando a:



**Ilustración 21: Modelo nodal de los *Child Process***

Las componentes que modifican los parámetros AIMD se encuentran en el *Function Block* que es donde se trabaja.

Creando una variable que guarde la ventana de congestión predicha para un peso de  $\frac{1}{n}$  y algoritmo de evasión de congestión Reno la simulación entrega los siguientes resultados.



**Ilustración 22: Ventana predicha encontrada para algoritmo Reno en la simulación**

## 3.5 Implementación entorno Linux

El sistema operativo Linux posee la ventaja de ser de código abierto y por lo tanto es posible manipular a voluntad del usuario componentes muy específicos siempre y cuando se tenga el dominio computacional necesario.

Particularmente en el presente proyecto se tiene la característica que el código encargado de la evasión de congestión corresponde a un módulo conectable (*pluggable congestion avoidance modules*), es decir que mediante una línea de comando es posible elegir qué control de congestión a utilizar. Por lo anterior la complejidad del proyecto recae en la programación del nuevo control de congestión.

A continuación se presenta la manera en que este procedimiento se realiza para así entregar los pasos necesarios a seguir en caso de necesitar repetir la experiencia. Todos los comandos que necesiten ser ingresados en la terminal deben ser ejecutados como super usuario (*sudo*).

### 3.5.1 Entorno de pruebas

Para este proyecto las pruebas corresponden a simulaciones de transferencias de datos sin información entre dos computadores adyacentes enlazados por una conexión Ethernet de 1[Gbps]. Dado la cercanía de los equipos la transferencia de datos es bastante veloz pero no resulta útil para observar el comportamiento del módulo, por ello se simulan distintos parámetros de *RTT* y tasa de pérdidas en la red para así representar casos de distancia variable en los equipos, niveles distintos de congestión en la red, etc.

Uno de los equipos corresponde al servidor donde trabaja el módulo de evasión de congestión y se simula los parámetros de la red. Los datos son recibidos en el equipo cliente.

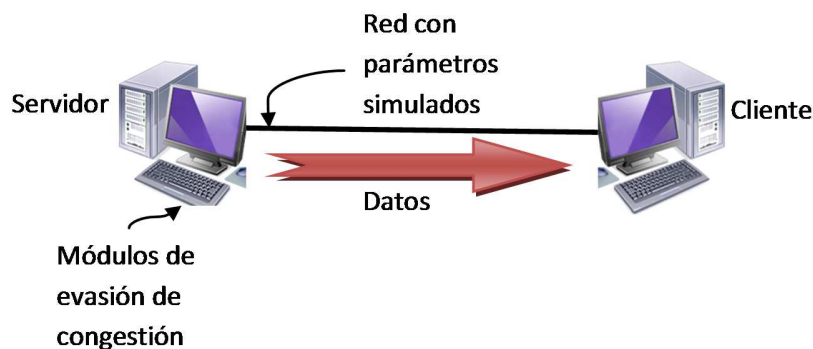


Ilustración 23: Esquema de conexión del entorno de pruebas

## 3.5.2 Pasos previos

A continuación se presentan las etapas necesarias de configuración del sistema operativo Linux.

Se requiere de una serie de etapas para llegar a la alteración y compilación de módulos. El primer paso consiste en dejar el sistema Linux con la capacidad de crear un módulo de evasión de congestión sin tener que compilar el *kernel* completo para cada modificación. Para ello primero se requiere descargar mediante algún gestor de descargas de la distribución Linux (se utilizó el gestor de paquetes *Synaptic*) las fuentes del *kernel* y los *headers*.

Con conexión a Internet en la consola se escribe lo siguiente para obtener los *headers*:

```
sudo apt-get install linux-headers-$(uname -r) linux-libc-dev kernel-pACKage
```

Luego hay que descargar el código de fuente del *kernel* utilizando el gestor de descargas *Synaptic*, en caso de que *Synaptic* no se encuentre en el sistema operativo se descarga del gestor que viene incluido en la distribución Linux correspondiente.

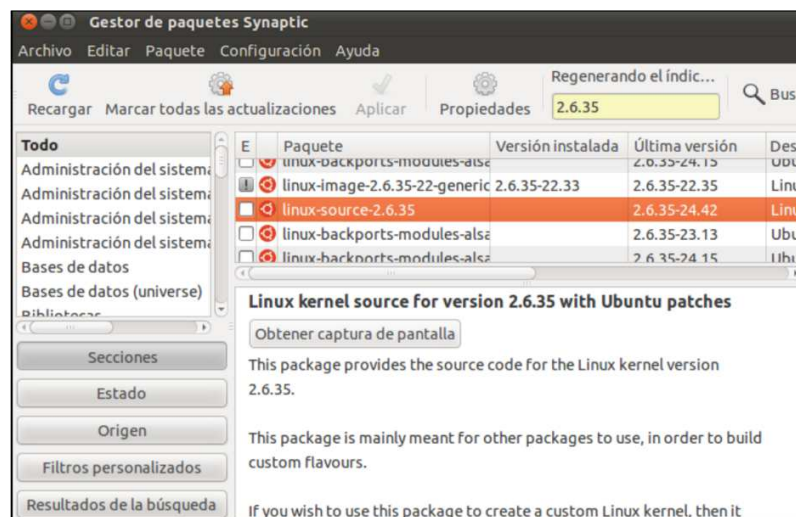


Ilustración 24: Fuentes y cabeceras del kernel de Linux obtenidas con Synaptic

Con esto tanto los *headers* como el *kernel* quedan en la dirección `/usr/src/Linux-source-2.6.35`. Por defecto el código está comprimido y se descomprime escribiendo el siguiente comando en la terminal:

```
sudo tar jvxf linux-source-2.6.35.tar.bz2
```

Lo siguiente consiste en preparar al sistema operativo para trabajar a través de módulos. Para ello se utiliza la herramienta *module-assistant*, descargándola como sigue:

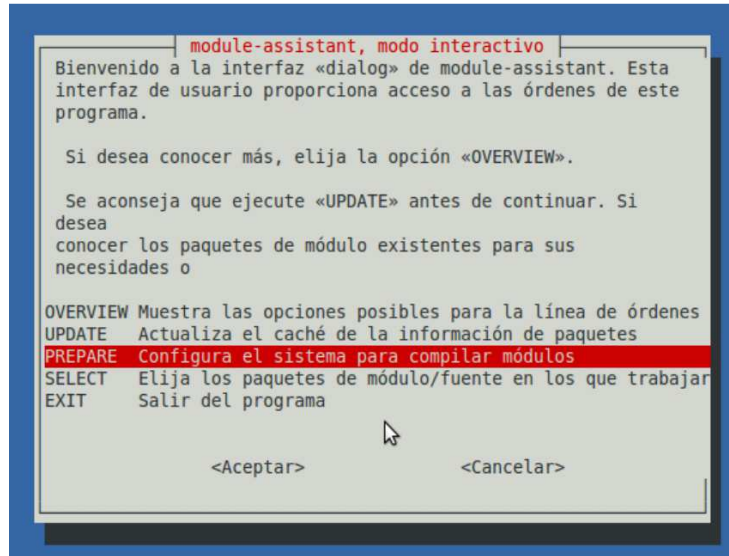
```
sudo apt-get install module-assistant
```



Tras ejecutarlo:

```
sudo module-assistant
```

Se obtiene lo siguiente en donde hay que ejecutar la opción PREPARE que dejara a Linux con la capacidad de compilar módulos, descargando el *built-essential* y escribiendo en */usr/src/linux*.



**Ilustración 25: Module-assistant para dejar al sistema operativo con la capacidad de compilar módulos.**

La carpeta */usr/src/linux-source-2.6.35/linux-source-2.6.35/net/ipv4* queda con las fuentes de los módulos de control de congestión, son visibles por ejemplo los módulos BIC, Cubic, Vegas, etc. En esta carpeta se guarda la fuente del módulo a implementar. Sin embargo para módulos nuevos que no vengan en la incorporación por defecto del sistema operativo es necesario modificar el archivo *Makefile* de la misma carpeta. Se muestra el *Makefile* original:

```
# Makefile for the Linux TCP/IP (INET) layer.
#
obj-y      := route.o inetpeer.o protocol.o \
             ip_input.o ip_fragment.o ip_forward.o ip_options.o \
             ip_output.o ip_sockglue.o inet_hashtables.o \
             inet_timewait_sock.o inet_connection_sock.o \
             tcp.o tcp_input.o tcp_output.o tcp_timer.o tcp_ipv4.o \
             tcp_minisocks.o tcp_cong.o \
             datagram.o raw.o udp.o udplite.o \
             arp.o icmp.o devinet.o af_inet.o igmp.o \
             fib_frontend.o fib_semantics.o \
             inet_fragment.o

obj-$(CONFIG_SYSCTL) += sysctl_net_ipv4.o
obj-$(CONFIG_IP_FIB_HASH) += fib_hash.o
obj-$(CONFIG_IP_FIB_TRIE) += fib_trie.o
obj-$(CONFIG_PROC_FS) += proc.o
obj-$(CONFIG_IP_MULTIPLE_TABLES) += fib_rules.o
obj-$(CONFIG_IP_MROUTE) += ipmr.o
obj-$(CONFIG_NET_IPIP) += ipip.o
obj-$(CONFIG_NET_IPGRE) += ip_gre.o
obj-$(CONFIG_SYN_COOKIES) += syncookies.o
```

```

obj-$(CONFIG_INET_AH) += ah4.o
obj-$(CONFIG_INET_ESP) += esp4.o
obj-$(CONFIG_INET_IPCOMP) += ipcomp.o
obj-$(CONFIG_INET_XFRM_TUNNEL) += xfrm4_tunnel.o
obj-$(CONFIG_INET_XFRM_MODE_BEET) += xfrm4_mode_beet.o
obj-$(CONFIG_INET_LRO) += inet_lro.o
obj-$(CONFIG_INET_TUNNEL) += tunnel4.o
obj-$(CONFIG_INET_XFRM_MODE_TRANSPORT) += xfrm4_mode_transport.o
obj-$(CONFIG_INET_XFRM_MODE_TUNNEL) += xfrm4_mode_tunnel.o
obj-$(CONFIG_IP_PNP) += ipconfig.o
obj-$(CONFIG_NETFILTER) += netfilter.o netfilter/
obj-$(CONFIG_INET_DIAG) += inet_diag.o
obj-$(CONFIG_INET_TCP_DIAG) += tcp_diag.o
obj-$(CONFIG_NET_TCPPROBE) += tcp_probe.o
obj-$(CONFIG_TCP_CONG_BIC) += tcp_bic.o
obj-$(CONFIG_TCP_CONG_CUBIC) += tcp_cubic.o
obj-$(CONFIG_TCP_CONG_WESTWOOD) += tcp_westwood.o
obj-$(CONFIG_TCP_CONG_HSTCP) += tcp_highspeed.o
obj-$(CONFIG_TCP_CONG_HYBLA) += tcp_hybla.o
obj-$(CONFIG_TCP_CONG_HTCP) += tcp_htcp.o
obj-$(CONFIG_TCP_CONG_VEGAS) += tcp_vegas.o
obj-$(CONFIG_TCP_CONG_VENO) += tcp_veno.o
obj-$(CONFIG_TCP_CONG_SCALABLE) += tcp_scalable.o
obj-$(CONFIG_TCP_CONG_LP) += tcp_lp.o
obj-$(CONFIG_TCP_CONG_YEAH) += tcp_yeah.o
obj-$(CONFIG_TCP_CONG_ILLINOIS) += tcp_illinois.o
obj-$(CONFIG_NETLABEL) += cipso_ipv4.o

obj-$(CONFIG_XFRM) += xfrm4_policy.o xfrm4_state.o xfrm4_input.o \
xfrm4_output.o

```

No es posible agregar el nuevo módulo directamente en el *Makefile* pues existe una lista predefinida de los archivos previamente mostrados en algún lugar de Linux, para agregar un nuevo módulo, por ejemplo *tcp\_test.c*, es necesario reemplazar el archivo *.o* del *Makefile* de alguno de la lista que no se utilice, aunque para ser más precisos al ejecutar el *Makefile* se compilan los archivos listados pero en este caso ya lo están, solo se estaría compilando el nuevo módulo, entonces en realidad puede reemplazarse cualquier archivo de los mostrados y solo en caso de que se modifiquen (que no será el caso del presente proyecto) se necesitara una compilación, en este caso se marca *tcp\_yeah.o* en color amarillo y se reemplazara por el módulo de ejemplo *tcp\_test.c*:

```

# Makefile for the Linux TCP/IP (INET) layer.
#

obj-y      := route.o inetpeer.o protocol.o \
ip_input.o ip_fragment.o ip_forward.o ip_options.o \
ip_output.o ip_sockglue.o inet_hashtables.o \
inet_timewait_sock.o inet_connection_sock.o \
tcp.o tcp_input.o tcp_output.o tcp_timer.o tcp_ipv4.o \
tcp_minisocks.o tcp_cong.o \
datagram.o raw.o udp.o udplite.o \
arp.o icmp.o devinet.o af_inet.o igmp.o \
fib_frontend.o fib_semantics.o \
inet_fragment.o

obj-$(CONFIG_SYSCTL) += sysctl_net_ipv4.o
obj-$(CONFIG_IP_FIB_HASH) += fib_hash.o
obj-$(CONFIG_IP_FIB_TRIE) += fib_trie.o
obj-$(CONFIG_PROC_FS) += proc.o
obj-$(CONFIG_IP_MULTIPLE_TABLES) += fib_rules.o
obj-$(CONFIG_IP_MROUTE) += ipmr.o
obj-$(CONFIG_NET_IPIP) += ipip.o
obj-$(CONFIG_NET_IPGRE) += ip_gre.o

```

```

obj-$(CONFIG_SYN_COOKIES) += syncookies.o
obj-$(CONFIG_INET_AH) += ah4.o
obj-$(CONFIG_INET_ESP) += esp4.o
obj-$(CONFIG_INET_IPCOMP) += ipcomp.o
obj-$(CONFIG_INET_XFRM_TUNNEL) += xfrm4_tunnel.o
obj-$(CONFIG_INET_XFRM_MODE_BEET) += xfrm4_mode_beet.o
obj-$(CONFIG_INET_LRO) += inet_lro.o
obj-$(CONFIG_INET_TUNNEL) += tunnel4.o
obj-$(CONFIG_INET_XFRM_MODE_TRANSPORT) += xfrm4_mode_transport.o
obj-$(CONFIG_INET_XFRM_MODE_TUNNEL) += xfrm4_mode_tunnel.o
obj-$(CONFIG_IP_PNP) += ipconfig.o
obj-$(CONFIG_NETFILTER) += netfilter.o netfilter/
obj-$(CONFIG_INET_DIAG) += inet_diag.o
obj-$(CONFIG_INET_TCP_DIAG) += tcp_diag.o
obj-$(CONFIG_NET_TCPPROBE) += tcp_probe.o
obj-$(CONFIG_TCP_CONG_BIC) += tcp_bic.o
obj-$(CONFIG_TCP_CONG_CUBIC) += tcp_cubic.o
obj-$(CONFIG_TCP_CONG_WESTWOOD) += tcp_westwood.o
obj-$(CONFIG_TCP_CONG_HSTCP) += tcp_highspeed.o
obj-$(CONFIG_TCP_CONG_HYBLA) += tcp_hybla.o
obj-$(CONFIG_TCP_CONG_HTCP) += tcp_htcp.o
obj-$(CONFIG_TCP_CONG_VEGAS) += tcp_vegas.o
obj-$(CONFIG_TCP_CONG_VENO) += tcp_veno.o
obj-$(CONFIG_TCP_CONG_SCALABLE) += tcp_scalable.o
obj-$(CONFIG_TCP_CONG_LP) += tcp_lp.o
obj-$(CONFIG_TCP_CONG_YEAH) += tcp_test.o
obj-$(CONFIG_TCP_CONG_ILLINOIS) += tcp_illinois.o
obj-$(CONFIG_NETLABEL) += cipso_ipv4.o

obj-$(CONFIG_XFRM) += xfrm4_policy.o xfrm4_state.o xfrm4_input.o \
xfrm4_output.o

```

Faltaría una última modificación al *Makefile* para indicarle al comando *make* con que compilara, estas líneas se agregan al final del código:

```

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

```

Con esto, escribiendo *make* en el terminal en esta carpeta se compilara el modulo deseado creando un archivo *tcp\_test.o* y *tcp\_test.ko* a partir del archivo fuente *tcp\_test.c*.

### 3.5.3 Estructura de TCP en Linux

En base a una breve descripción de los distintos archivos componentes de *TCP* en Linux se presentara la estructura de un módulo evasión de congestión típico, el código final está en los anexos del proyecto.

Como se ha especificado anteriormente el componente de control de congestión corresponde a solo una fracción de lo abarcado por *TCP*, particularmente en Linux, *TCP* posee distintos programas encargados del correcto funcionamiento de la transmisión de datos y se puede separar sus archivos constituyentes de acuerdo a su objetivo [1]:

- *tcp.h*: Incluye las definiciones relacionadas TCP, incluidas las estructuras de datos definidas anteriormente. Este archivo existe tanto en `include / net /` y `include / linux / directorios`.
- *tcp.c*: Incluye el código de TCP general y cubre la interfaz entre diferentes sockets y el resto del código TCP.

- `tcp_input.c`: Archivo más grande e importante encargado de los paquetes entrantes de la red. También contiene el código para la máquina en estado de recuperación.
- `tcp_output.c`: Encargado de los paquetes salientes a la red. Contiene algunas de las funciones que son llamadas desde el sistema de control de la congestión.
- `tcp_ipv4.c`: Código específico de TCP IPv4. Esta función entrega los paquetes relevantes al framework de control de congestión.
- `tcp_timer.c`: Implementa funciones de administración de temporizadores.
- `tcp_cong.c`: Implementa el soporte de control de congestión conectable (*pluggable*) de TCP.
- `tcp_[name of algorithm].c`: Algoritmos de evasión de congestión específicos a utilizar. El objetivo del proyecto corresponde a esta área.

La manera en que *TCP* maneja la interfaz de los distintos algoritmos de evasión de congestión se describe en `struct_tcp_congestion_ops` definida en el archivo `tcp.h`.

```
struct tcp_congestion_ops {
    struct list_head list
    unsigned long flags
    void(* init )(struct sock *sk)
    void(* release )(struct sock *sk)
    u32(* ssthresh )(struct sock *sk)
    u32(* min_cwnd )(const struct sock *sk)
    void(*cong_avoid )(struct sock *sk, u32 ACK, u32 in_flight)
    void(*set_state )(struct sock *sk, u8 new_state)
    void(*cwnd_event )(struct sock *sk, enum tcp_ca_event ev)
    u32(* undo_cwnd )(struct sock *sk)
    void(*pkts_ACKed )(struct sock *sk, u32 num_ACKed, s32 rtt_us)
    void(*get_info )(struct sock *sk, u32 ext, struct sk_buff *skb)
    char name [TCP_CA_NAME_MAX]
    struct module *owner
}
```

De los cuales los métodos `ssthresh`, `cong_avoid` son requeridos obligatoriamente y cuyos objetivos son:

- `ssthresh()`: Calcula el umbral del *slow-start*, cuando la ventana de congestión está bajo dicho umbral la conexión esta en modo *slow-start* en vez del modo de evasión de congestión. Este método se ejecuta cuando ocurre una pérdida igualando la ventana de congestión al valor del *ssthresh* correspondiendo a la ventana multiplicada por  $\beta$ .
- `cong_avoid()`: Este método se invoca cuando se recibe un *ACK* y puede responder incrementando la ventana de congestión realizando una nueva calculación de esta de acuerdo al algoritmo particular en uso (*Reno*, *Vegas* o *Cubic entre otros*)

Luego la estructura del código es del tipo:

```
Struct data{
//variables usadas:  $\alpha$ ,  $\beta$ , weight, contadores.
}
```

```

Init{
//inicialización de variables
}
Cong avoid{
//cálculo de nuevo valor de la ventana de congestión creciente. El incremento aditivo ocurre
en esta sección.
}
Ssthresh{
//método que entrega el valor de ssthresh para reducir la ventana en dicho valor. El
decremento multiplicativo no ocurre en esta sección, sino que prácticamente se entrega
indirectamente el beta asociado y el decremento ocurre en otra parte del sistema operativo en
base a ello.
}

```

Aunque ambos códigos, módulo de Linux y simulación OPNET, están escritos en lenguaje C, existe una diferencia que limita cierta precisión: en los módulos de evasión de congestión las variables deben ser únicamente enteras y por lo tanto para números racionales se necesitan dos variables: la variable no normalizada y el normalizador; ambas del tipo entero. Ejemplo:  $\alpha = \frac{\alpha_{int}}{\alpha_{norm}}$  con  $\alpha = 0.7$  y  $\alpha_{norm} = 1000$  indica que  $\alpha_{int} = 700$ . Lo importante es dejar las variables normalizadoras al final de los cálculos para evitar así cualquier error de truncación.

Como se aprecia en el código final del módulo existen diferencias respecto al planteamiento teórico original (incluyendo simulación en OPNET). Al sistema operativo le molesta los instantes en donde la ventana de congestión permanece inalterada, esto ocurre cuando  $\alpha=0$  o  $\beta=1$  y cuando se presenta dicho caso ocurren comportamientos que alteran la ventana de congestión de manera totalmente independiente al módulo de control montado. Como solución se utiliza el mayor  $\beta$  posible menor que 1 el cual resulta ser 0.9 (considerando que se trabaja con variables enteras), con este valor de  $\beta$  y la relación de *fairness* se obtiene el valor de  $\alpha$  mínimo mayor que 0 que corresponde ser 0.16. Con los cambios planteados ya no será posible llegar a la ventana predicha y permanecer invariante en dicho valor, se presentarán oscilaciones que en su medida debiesen ser mucho menos pronunciadas que las producidas por otros algoritmos de evasión de congestión como *Reno* por ejemplo.

En base a ello los valores de  $\alpha$  y  $\beta$  quedan para el método de diferencia porcentual:

$$\alpha = \max \left\{ \frac{(cwnd_{pred} - cwnd)}{cwnd_{pred}}, 0.16 \right\}$$

$$\beta = \min \left\{ \beta_F, \frac{cwnd_{pred}}{cwnd}, 0.9 \right\}$$

Y para el controlador PI

$$\alpha = \max \left\{ K_P e(t) + K_I \sum e[n], 0.16 \right\}$$

$$\beta = \min \left\{ \beta_F, \frac{cwnd_{pred}}{cwnd}, 0.9 \right\}$$

### 3.5.4 Software requerido

Además del módulo operativo se necesita software específico orientado a crear un entorno de pruebas y medir el rendimiento del sistema de control de congestión. Las aplicaciones a utilizar corresponden a *NetEm*, *Iperf*, *tcpprobe* y *MATLAB*.

#### 3.5.4.1 NetEm

NetEm corresponde a un software que provee funcionalidad de emulación de red cuyo objetivo principal para el proyecto es añadir, entre otros parámetros, *RTT* y tasa de pérdida a una conexión de una tarjeta de red específica del sistema (sea tanto virtual, *Ethernet* o *Wi-Fi*).

Este software se aplica en la tarjeta red del equipo servidor pues el *RTT* es indiferente si se aplica a paquetes de ida o de regreso, sin embargo para la pérdida no ocurre lo mismo y tiene más sentido que se pierdan paquetes de datos en vez de los acuse de recibo por parte del cliente aunque no hay que perder la noción de que cualquier paquete puede perderse dado la pérdida intrínseca propia de la red.

Esta aplicación se utiliza mediante terminal de manera que previa a la inicialización de una prueba de transferencia se asignan los valores de *RTT* y tasa de pérdida con el comando:

```
tc qdisc add dev eth0 root netem delay 100ms loss 1%
```

Donde *eth0* corresponde a la interfaz de la tarjeta de red a la cual se aplica esta regla (puede tener otros nombres de acuerdo a la cantidad de tarjetas de red en el equipo: *eth1*, *eth2*), *delay* al *RTT* expresado en milisegundos y *loss* a la tasa de pérdida de paquetes expresado en porcentaje.

#### 3.5.4.2 Iperf

*Iperf* corresponde a la aplicación encargada de transmitir flujo de datos por una red determinada definida por un equipo servidor y un equipo cliente. La aplicación más típica de este software es la evaluación del rendimiento de una red específica. Los datos generados por esta aplicación son basura y por defecto no se almacenan en el disco duro sino que se elimina post recepción, esto produce que la medición no se vea afectada por factores externos de la conexión como por ejemplo en la velocidad de escritura del disco duro como sería el caso de utilizar una transmisión ftp para el cometido.

Entre las opciones disponibles entregadas por *Iperf* está el tipo de transferencia (*TCP* o *UDP*), tamaño de los datagramas, número de conexiones simultáneas, valor del *TTL* (*time to*

*live*). Para el proyecto se utilizan datos *TCP* y el resto por defecto pues para las pruebas no se requiere una configuración muy específica y se busca tener resultados bastantes generales.

Esta aplicación mide el *throughput* en intervalos de tiempo definido y al final de la transmisión entrega el promedio. Los valores son entregados numéricamente y sirven como se mencionó previamente para medir el rendimiento de la red, sin embargo para los propósitos del presente proyecto resultan ser insuficientes: más que obtener numéricamente los valores de *throughput*, es de mucha más utilidad para el análisis del módulo y evaluación de su comportamiento tener la evolución de la ventana de congestión en el tiempo. Esto es porque aunque tengamos una medición de la tasa de transferencia no es posible verificar que la convergencia esté funcionando correctamente: es posible que numéricamente los valores estén correctos pero el módulo podría estar actuando erróneamente sin converger y en promedio (datos entregados por *Iperf*) coincidan. Por esto se utilizara este software con el propósito de transmitir datos y sus resultados serán utilizados como una mera referencia. Se utilizara otro software para verificar el comportamiento de la ventana de congestión y con ello el funcionamiento del módulo.

*Iperf* también se utiliza por comandos en la terminal y es una aplicación del tipo cliente-servidor, es decir que para el entorno de pruebas se requerirá que el software este presente tanto en el equipo servidor como en el equipo cliente. Hay que hacer notar que para propósitos del proyecto el equipo asociado a la descarga de datos es el cliente y el asociado a subida corresponde al servidor, características opuestas en *Iperf* en donde el servidor es el asociado a descarga y el cliente el asociado a la subida de datos. Por lo anterior para no causar confusión se utilizaran las definiciones de *Iperf* solo durante esta sección orientada a este programa.

En el servidor hay que colocar el comando en la terminal:

```
iperf -s
```

Con esto el servidor quedara en escucha esperando al cliente que le envíe los datos. En el cliente hay que colocar:

```
iperf -c <dirección IP del servidor> -t <tiempo de la prueba en segundos> -i <tiempo de intervalos de muestra de resultados en segundos>
```

Por ejemplo: `iperf -c 192.168.5.4 -t 30 -i 5` Se conecta al servidor 192.168.5.4, la prueba dura 30 segundos y muestra los datos resultantes cada 5 segundos.

Los datos entregados por *Iperf* corresponden a los datos de intervalo temporal, datos transferidos y tasa de transferencia durante dicho intervalo, por ejemplo:

```
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[  4] local 192.168.5.4 port 5001 connected with 192.168.5.226 port
46181
[ ID] Interval      Transfer      Bandwidth
[  4] 0.0- 5.0 sec   56.3 MBytes   94.5 Mbits/sec
[  4] 5.0-10.0 sec   56.3 MBytes   94.5 Mbits/sec
```

```

[ 4] 10.0-15.0 sec 56.3 MBytes 94.5 Mbits/sec
[ 4] 15.0-20.0 sec 56.3 MBytes 94.5 Mbits/sec
[ 4] 20.0-25.0 sec 56.3 MBytes 94.5 Mbits/sec

```

### 3.5.4.3 TCP\_probe

*TCP\_probe* es un pequeño software encargado de realizar un *probing* a la red únicamente de datos *TCP*. Guarda tanto el tiempo, tamaño de la ventana de congestión, el *ssthresh* y muchos otros parámetros innecesarios para el proyecto. Adicionalmente, dado que se trata de software de código abierto, se realizó una modificación en donde se agrega entre los parámetros que guarda la ventana de congestión predicha. Esto es para, una vez con los datos guardados de una prueba, realizar los gráficos correspondientes.

La modificación trata de dos partes, la primera agregar una variable nueva llamada *cwnd\_pred* en la librería *tcp.h* (esta librería se encuentra en varios directorios del sistema operativo, se recomienda modificar cada uno). En la línea 327 comienzan las declaraciones de variables utilizadas por *tcp\_probe*, esa sección de *tcp.h* debe quedar así:

```

u32     snd_ssthresh; /* Slow start size threshold */
u32     snd_cwnd;     /* Sending congestion window */
u32     snd_cwnd_cnt; /* Linear increase counter */
u32     snd_cwnd_clamp; /* Do not allow snd_cwnd to grow above this */
u32     snd_cwnd_used;
u32     snd_cwnd_stamp;
u32     snd_cwnd_pred; /* Variable agregada para el proyecto de memoria*/

```

La segunda modificación corresponde en el código *tcp\_probe.c*, consta de tres cambios en el código, se muestran las alteraciones marcadas en amarillo:

Crear variable dentro del código:

```

struct tcp_log {
    ktime_t tstamp;
    __be32  saddr, daddr;
    __be16  sport, dport;
    u16     length;
    u32     snd_nxt;
    u32     snd_una;
    u32     snd_wnd;
    u32     snd_cwnd;
    u32     ssthresh;
    u32     srtt;
    u32     snd_cwnd_pred;
}

```

Asignarle a la variable de *tcp.h* cuyo valor es el que se actualiza acorde al funcionamiento del módulo:

```

p->tstamp = ktime_get();
p->saddr = inet->inet_saddr;
p->sport = inet->inet_sport;
p->daddr = inet->inet_daddr;
p->dport = inet->inet_dport;
p->length = skb->len;
p->snd_nxt = tp->snd_nxt;
p->snd_una = tp->snd_una;

```



```

p->snd_cwnd = tp->snd_cwnd;
p->snd_wnd = tp->snd_wnd;
p->sssthresh = tcp_current_ssthresh(sk);
p->srtt = tp->srtt >> 3;
p->snd_cwnd_pred=tp->snd_cwnd_pred;

```

Agregar la variable a la impresión, ahora una columna de los datos corresponde a `cwnd_pred`:

```

return scnprintf(tbuf, n,"%lu.%09lu %pI4:%u %pI4:%u %d %#x %#x %u %u %u %u\n",
(unsigned long) tv.tv_sec,
(unsigned long) tv.tv_nsec,
&p->saddr, ntohs(p->sport),
&p->daddr, ntohs(p->dport),
p->length, p->snd_nxt, p->snd_una,
p->snd_cwnd, p->sssthresh, p->snd_cwnd_pred, p->snd_wnd, p->srtt);

```

*TCP\_probe* como lo son los algoritmos de evasión de congestión corresponde a un módulo que debe ser cargado para utilizarse.

Este programa se utiliza vía terminal de la siguiente manera:

```

modprobe tcp_probe port=5001
chmod 444 /proc/net/tcpprobe
cat /proc/net/tcpprobe >/tmp/tcpprobe.out &
TCPCAP=$!
iperf -i 10 -t 100 -c receiver
kill $TCPCAP

```

Este ejemplo registra los datos *TCP* salientes por el puerto 5001 de una corrida de *Iperf* (que efectivamente utiliza el puerto 5001) y guarda los resultados en un archivo de texto de nombre *tcpprobe.out* en la carpeta definida en el ejemplo:

```

6.649503840 10.10.10.20:33429 10.10.10.10:5001 32 0x5d2cd08 0x5ce4b28 204 204 183 1453248 2
6.649623866 10.10.10.20:33429 10.10.10.10:5001 32 0x5d2d858 0x5ce5678 204 204 183 1453248 2
6.649730398 10.10.10.20:33429 10.10.10.10:5001 32 0x5d2e3a8 0x5ce61c8 204 204 183 1453248 2
6.649856461 10.10.10.20:33429 10.10.10.10:5001 32 0x5d2fa48 0x5ce72c0 205 204 183 1453248 2
6.649971277 10.10.10.20:33429 10.10.10.10:5001 32 0x5d30b40 0x5ce83b8 205 204 183 1453248 2
6.650113098 10.10.10.20:33429 10.10.10.10:5001 32 0x5d31690 0x5ce8f08 205 204 183 1453248 2
6.650169456 10.10.10.20:33429 10.10.10.10:5001 32 0x5d32788 0x5cea000 205 204 183 1453248 2
6.650351676 10.10.10.20:33429 10.10.10.10:5001 32 0x5d332d8 0x5ceab50 205 204 183 1453248 2

```

En donde cada columna se refiere a un tipo de datos distinto y cada fila a un instante de tiempo. Las columnas importantes para el análisis de datos son la primera, la séptima, la octava y la novena que corresponden al tiempo en milisegundos, la ventana de congestión, el *sssthresh* y la ventana de congestión predicha respectivamente.

### 3.5.4.4 MATLAB®

*MATLAB*® es un poderoso software matemático con un lenguaje de programación propio que es de muy alto nivel por lo cual la escritura de programas resulta relativamente sencilla. Tiene una gran variedad de *toolbox* por lo cual es utilizado en varias ramas del conocimiento (control de sistemas, telecomunicaciones, redes neuronales, estadísticas y análisis numérico, procesamiento de señales entre otras aplicaciones.)

Para el presente proyecto de memoria se utiliza como un graficador de los resultados entregados por *tcp\_probe*. Dada la manera de entregar los datos por *tcp\_probe*, no es posible realizar un ploteo directo por lo tanto se escribe un programa llamado *tcpprobe\_reader\_any.m* que en base a un archivo de texto de *tcp\_probe* extrae la información necesaria y realiza un gráfico de *cwnd* y *cwnd\_prediction* versus el tiempo. Este programa es la base para el análisis de los resultados pues sus gráficos son los utilizados en la sección de análisis de resultados.

## 3.5.5 Procedimiento de operación del modulo

Ya con los programas requeridos en funcionamiento se procede a trabajarlos conjuntamente para realizar las pruebas correspondientes. Para ello se utilizan archivos *bash*, que prácticamente interpretan ordenes que incluyen la incorporación de los otros programas con el fin de simplificar el trabajo y así en vez de ejecutar cada programa independientemente y en el orden correcto basta con ejecutar el archivo *bash* deseado. Existe un archivo *bash* especial encargado de mantener los buffers de *TCP* en valores fijos para cada prueba, esto es para evitar que en alguna prueba en particular se alcance el límite del buffer y con ello la ventana de congestión se estanque y quede acotada por arriba por factores ajenos al módulo de evasión de congestión. Este *bash* se denomina *opciones\_buff* y esta descrito en los anexos.

De manera general el archivo *bash* a utilizar en las pruebas de es como sigue:

```
#cargar módulo tcp_probe
#cargar opciones_buff
#removev módulo de evasión de congestión actualmente montado (si no hay
ninguno esta línea la ignora y el resto del código sigue sin problemas)
#insertar el módulo de evasión de congestión a probar
#cargar el modulo insertado
#cargar netem con las condiciones de red deseadas
#cargar ipef
#copiar contenido de tcp_probe para ser guardado
#removev modulo
```

Se utiliza un archivo *bash* por modulo y se modifican los parámetros deseados. Por ejemplo el *bash* para una prueba del módulo que utiliza el controlador PI y con condiciones de red de un *RTT* de 10[ms] y una tasa de pérdidas de 0.1% el código es:

```
#! /bin/sh
```

```

#sudo insmod /usr/src/linux-source-3.2.0/linux-source-3.2.0/net/ipv4/tcp_probe.ko
sudo ./opciones_buff
sudo rmmod /usr/src/linux-source-3.2.0/linux-source-3.2.0/net/ipv4/tcp_conn_pi.ko
sleep 5
sudo insmod /usr/src/linux-source-3.2.0/linux-source-3.2.0/net/ipv4/tcp_conn_pi.ko
sudo sysctl -w net.ipv4.tcp_congestion_control=conn_pi
tc qdisc replace dev eth0 root netem delay 10ms loss 0.1%
./connect $1
cp /tmp/tcpprobe.out /owl/tcpprobe_conn_pi.info
sleep 5
sudo rmmod /usr/src/linux-source-3.2.0/linux-source-3.2.0/net/ipv4/tcp_conn_pi.ko

```

Los *sleep* presentes en el código son una prevención en caso de que alguna instrucción no sea instantánea y para que terminase sin problemas. El *./connect \$1* es un archivo *bash* que contiene la línea de *iperf* con la *ip* de destino y los intervalos de tiempo incluidos, el *\$1* corresponde el primer argumento de dicho *bash* e indica el tiempo de ejecución de *iperf* en segundos al momento de ejecutar el archivo.

Se concluye la etapa de implementación por lo que el sistema operativo y los equipos están en condiciones de realizar las pruebas necesarias para estudiar el comportamiento del módulo de evasión de congestión objetivo del proyecto de memoria.

# CAPÍTULO 4: ANÁLISIS DE RESULTADOS

## 4.1 Introducción

En este capítulo se presentan los resultados obtenidos de las pruebas realizadas. Estos datos son entregados separados de acuerdo a la variable medida en cada prueba y como ésta influye en el comportamiento de los módulos. Por comportamiento de los módulos se refiere principalmente a dos factores principales: la evolución de la ventana de congestión y la predicción de esta en el tiempo. Como métrica se utilizan las definiciones estadísticas de media y varianza de la ventana de congestión a partir del instante en donde el *slow-start* inicial termina hasta el final de la prueba y de la ventana de congestión predicha. También es posible realizar un análisis cualitativo en base a los resultados obtenidos indicando entre otros factores la relación entre la variable en cuestión y el comportamiento de los módulos (por ejemplo, indicar que bajo tal valor de la variable a ciertos parámetros la convergencia de la ventana de congestión predicha es lenta y por lo tanto la referencia para el controlador es inviable).

Las variables en cuestión utilizadas para el presente análisis corresponden a: *RTT*, tasa de pérdida, peso de las nuevas muestras para el cálculo de la predicción, tipo de peso a utilizar y constantes del controlador en el caso del módulo *tcp\_conn\_pi*. Para cada una de estas pruebas la variable en cuestión toma distintos valores mientras que las otras variables corresponden a parámetros fijos.

Los resultados son entregados tanto gráficamente como en tablas de datos. Los gráficos son en su mayoría de la ventana de congestión y la predicción versus el tiempo pero también se muestran otros que corresponden a datos estadísticos de media y desviación estándar para la ventana de congestión y la predicción.

También se entregan resultados del módulo Reno en donde no existirá un análisis propio sino más bien una comparativa con los módulos del presente proyecto.

Todas las siguientes pruebas tienen un peso de la ventana de congestión de  $\frac{1}{n}$  para la nueva estimación de la predicción a menos que se indique lo contrario y en el caso particular de módulo *tcp\_conn\_pi* los parámetros del controlador corresponden a  $K_P=10^{-5}$  y  $K_I=10^{-8}$  a excepción de la prueba en donde dichos parámetros son los evaluados. Las mediciones se realizaron en el Laboratorio de Telecomunicaciones ubicado en el segundo piso del edificio de Electrotecnologías.

## 4.2 Resultados de RTT variable

Para esta prueba no importa demasiado como varia el resultado de la ventana de congestión respecto a los distintos *RTT* aplicados, esto porque la forma en plantear el modulo deriva directamente de Reno en donde el *RTT* no influye en el tamaño de la ventana de congestión (*RTT* influye en el *throughput*), lo que importa es la comparación para un mismo *RTT* entre la ventana de congestión y la predicción de esta, pues esta relación indica de cierta manera el comportamiento del módulo en cuanto a su rendimiento (media y desviación estándar principalmente).

Las siguientes pruebas los valores utilizados de *RTT* son 10, 50 y 100 milisegundos, se utiliza un peso de  $\frac{1}{n}$  con  $n$  la cantidad de iteraciones para el cálculo de la referencia. Se utilizan dos parámetros distintos de tasa de pérdida: 0.1% y 0.001%.

Una prueba que analiza el *RTT* es importante mostrarla porque el *RTT* en si es una medida de distancia entre equipos, es decir un tiempo de retardo elevado indica que los equipos están físicamente distanciados, lo mismo para el caso de *RTTs* pequeños. Además, uno de los objetivos del módulo del proyecto corresponde a una solución alternativa de las redes *high bandwidth delay-product*, por lo tanto se necesita entender el comportamiento de este módulo bajo esas condiciones.

Se espera que el comportamiento de ambos módulos muestre el seguimiento de la ventana de congestión a la ventana predicha independiente del *RTT* aplicado. Además para que un resultado sea considerado bueno se requiere también que la desviación estándar de la ventana de congestión no sea elevada tomado la del módulo Reno como referencia comparativa.

## 4.2.1 Módulo tcp\_conn\_alt

Utilizando este módulo se busca estudiar cómo reacciona a los distintos retardos aplicados. Se espera por lo menos que la ventana de congestión se mantenga cerca de la referencia dado que este módulo utiliza un controlador proporcional porcentual.

### 4.2.1.1 Gráficos de ventana de congestión vs tiempo

1) Tasa de pérdida 0.1%

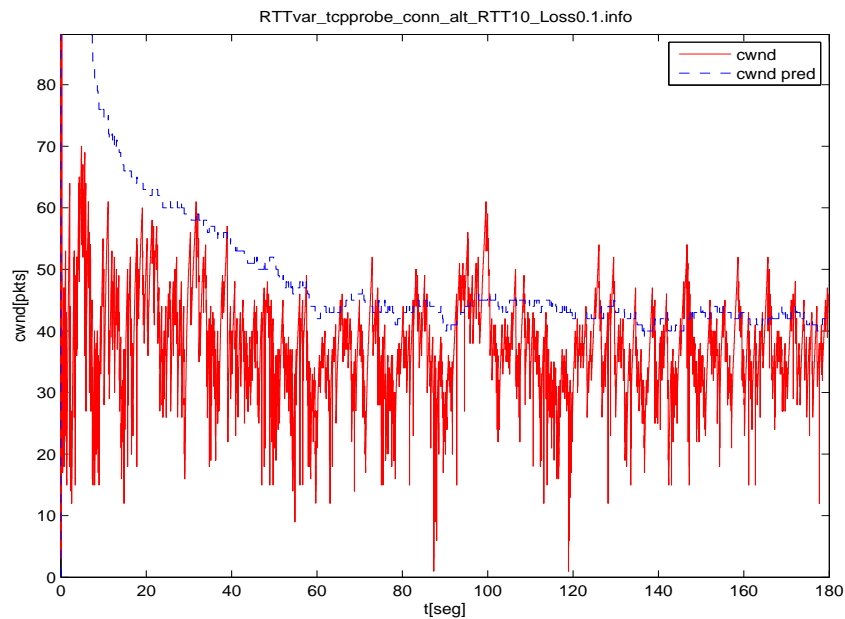


Ilustración 26: Módulo tcp\_conn\_alt RTT=10[ms] y Loss=0.1%

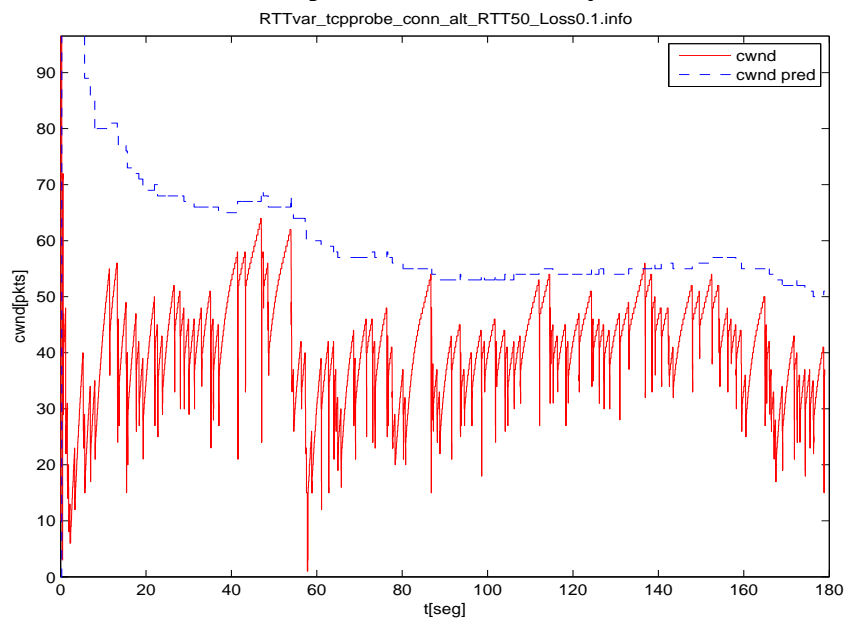
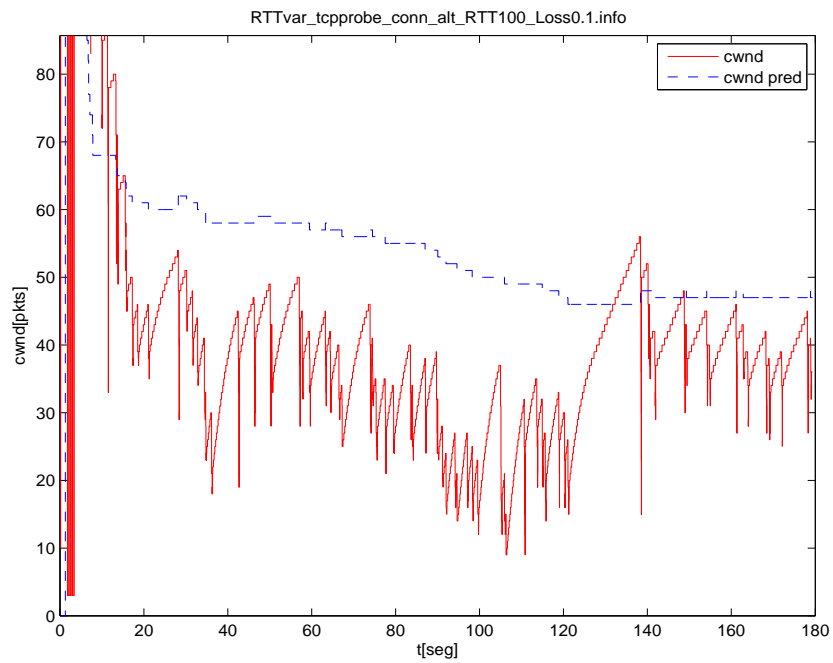
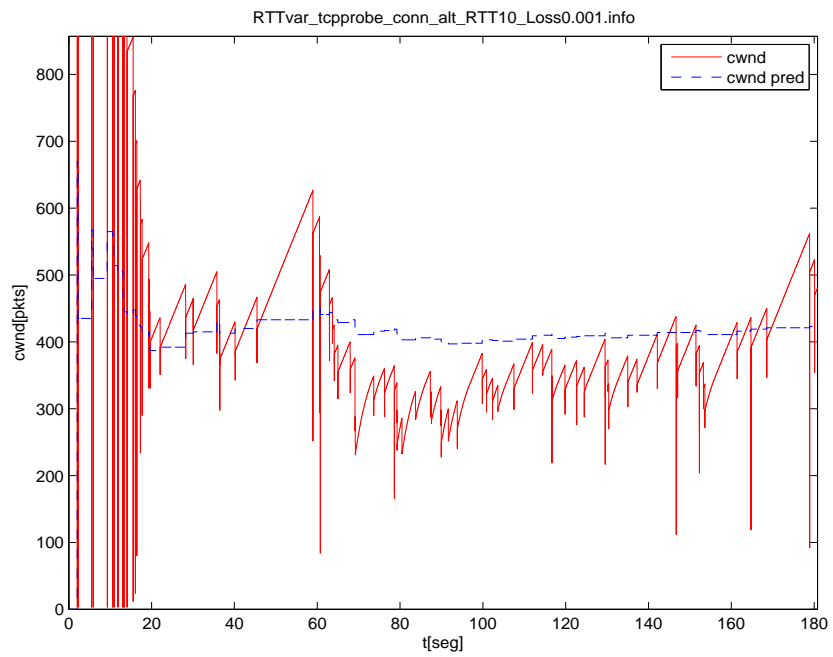


Ilustración 27: Módulo tcp\_conn\_alt RTT=50[ms] y Loss=0.1%

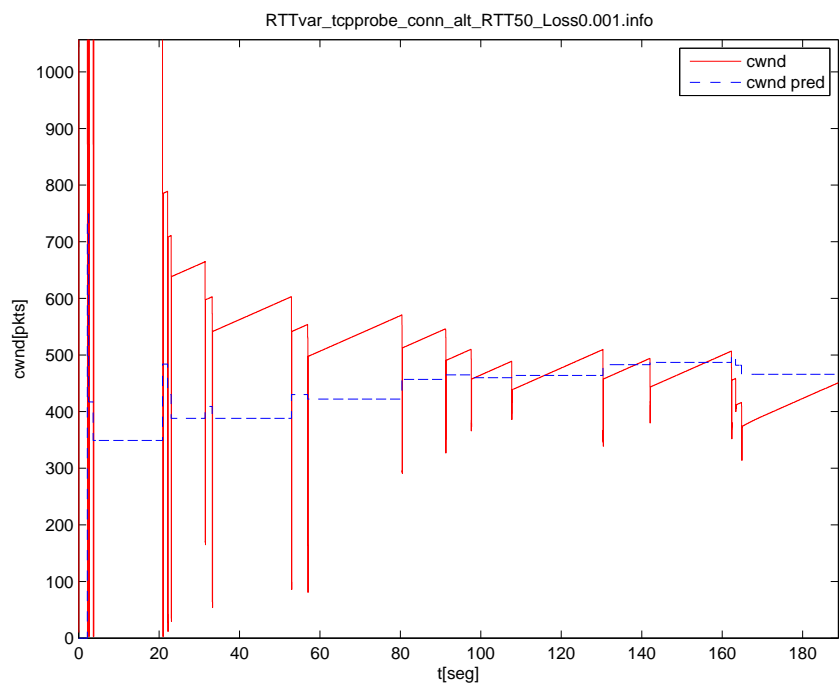


**Ilustración 28: Módulo tcp\_conn\_alt RTT=100[ms] y Loss=0.1%**

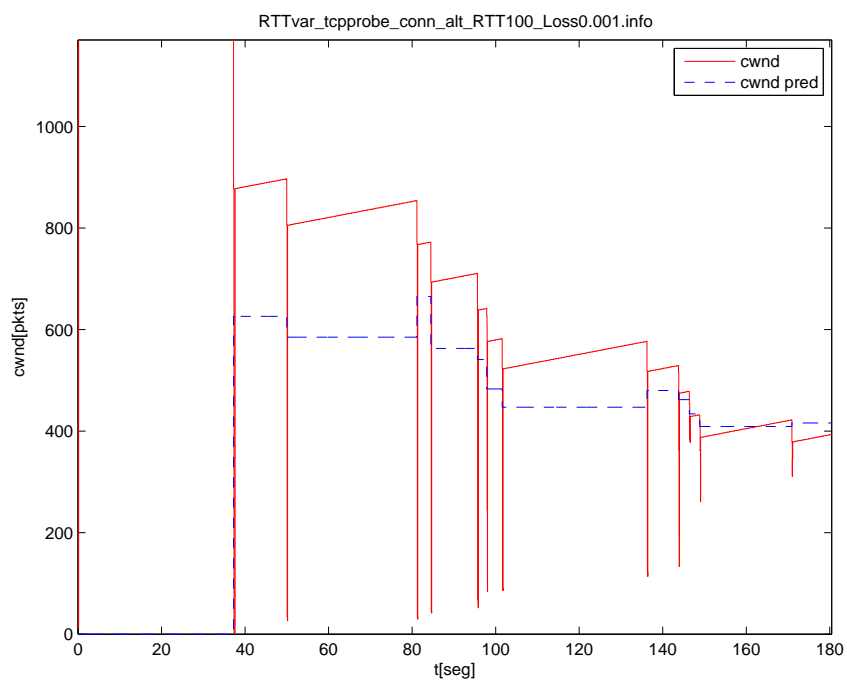
2) Tasa de pérdida 0.001%



**Ilustración 29: Módulo tcp\_conn\_alt RTT=10[ms] y Loss=0.001%**



**Ilustración 30: Módulo tcp\_conn\_alt RTT=50[ms] y Loss=0.001%**



**Ilustración 31: Módulo tcp\_conn\_alt RTT=100[ms] y Loss=0.001%**



Se aprecia que el comportamiento de este módulo está acorde a lo previsto. Dado que se trata de un controlador proporcional se contempla que la ventana de congestión permanece gran parte del tiempo bajo la referencia. Sin embargo este fenómeno ocurre más pronunciadamente a bajo *RTT*. A *RTT*s más altos los pulsos donde ocurre el crecimiento aditivo de la ventana antes de que ocurra una pérdida son anchos. Además parece ser que los valores de  $\alpha$  son invariantes cuando el *RTT* es elevado a diferencia de las pruebas de bajo *RTT* (10[ms]) en donde se aprecia la curvatura de la ventana dado que  $\alpha$  varía mientras no se alcanza la referencia.

Luego, el primer requisito para considerar el comportamiento del módulo como aceptable se cumple, falta verificar en cuanto a si las medias son parecidas y finalmente comprobar la desviación estándar presente.

#### 4.2.1.2 Resultados estadísticos para el módulo tcp\_conn\_alt con RTT variable

- 1) Tasa de pérdida de 0.1%

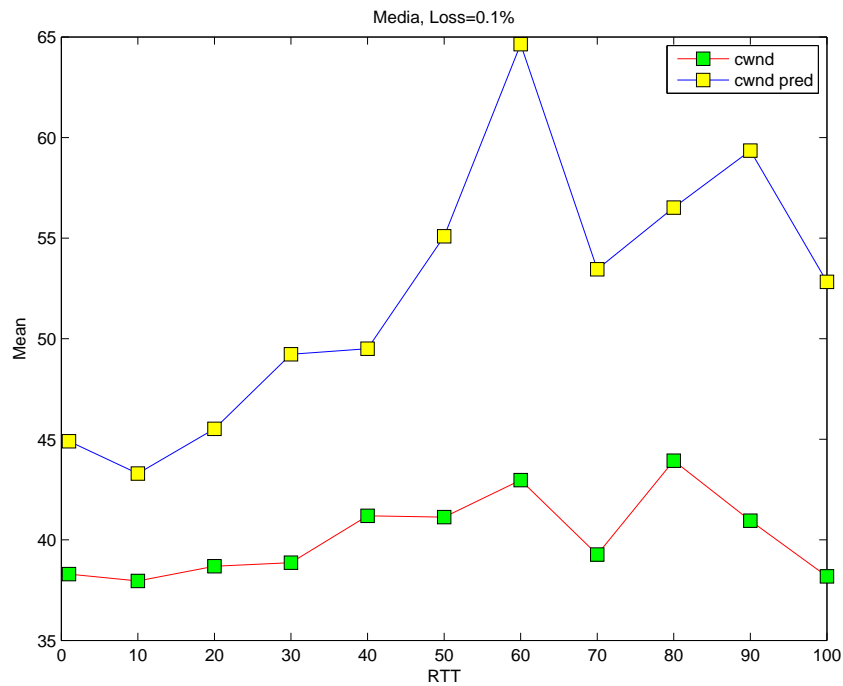
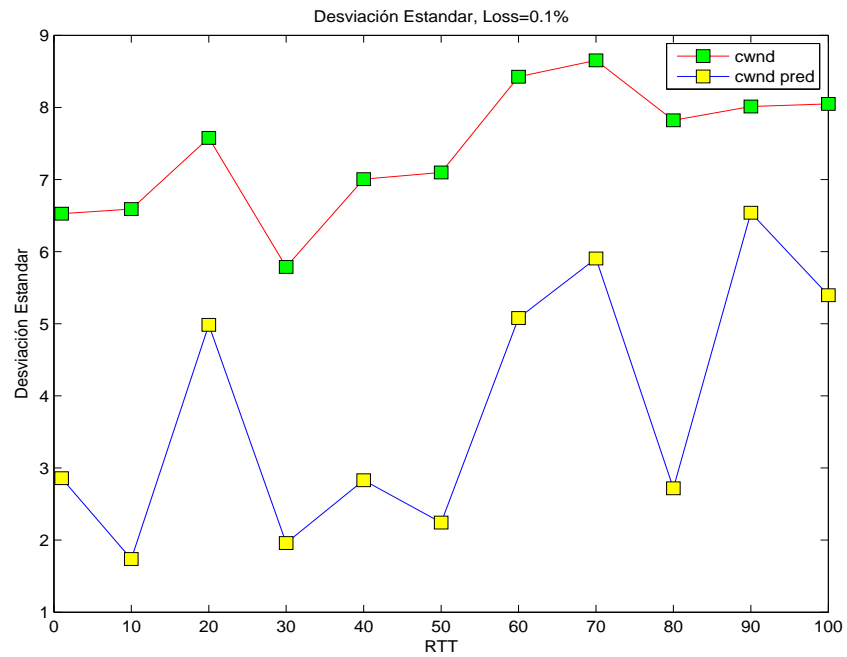
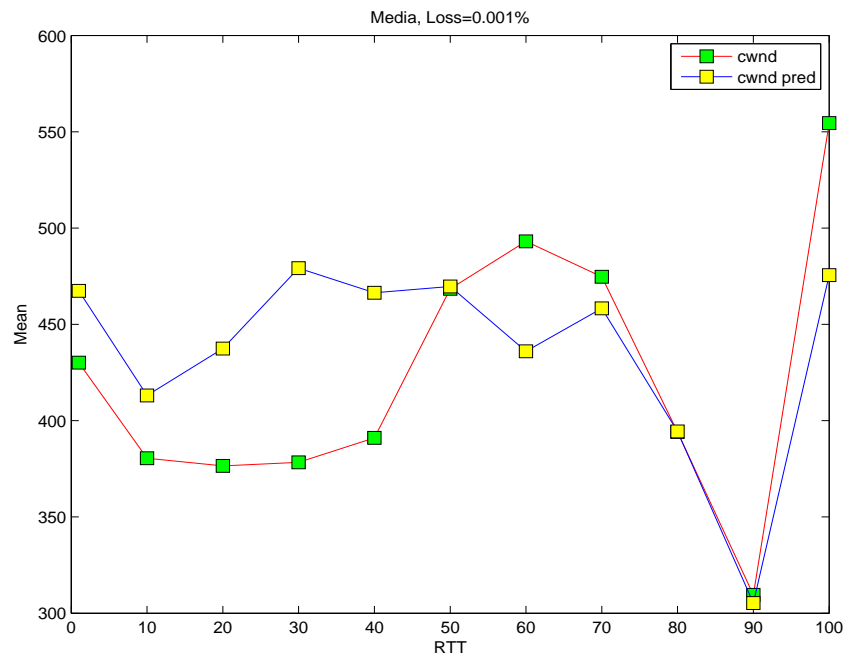


Ilustración 32: Media de la ventana de congestión para tcp\_conn\_alt para RTT variable con Loss=0.1%

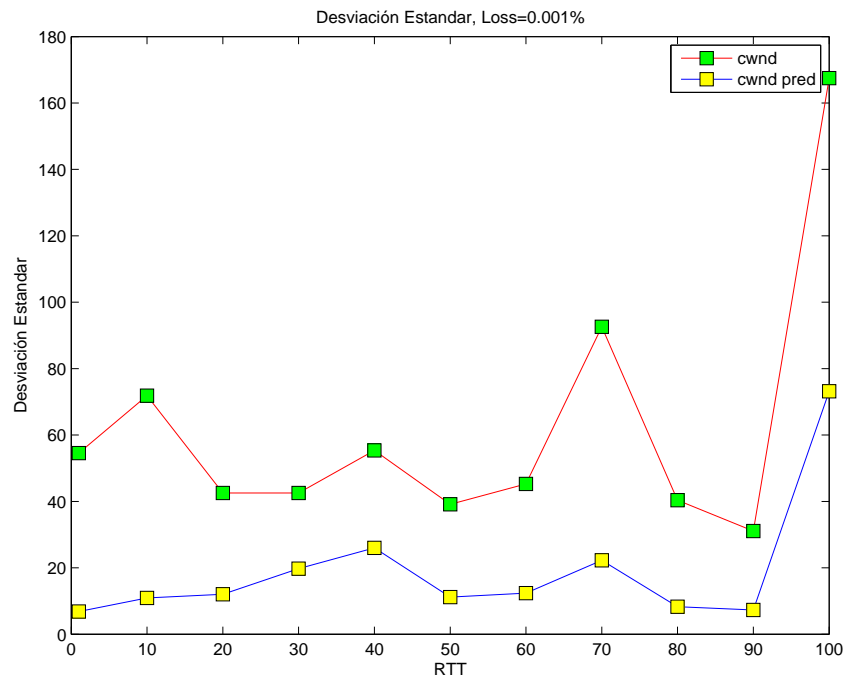


**Ilustración 33: Desviación estándar de la ventana de congestión para tcp\_conn\_alt para RTT variable con Loss=0.1%**

2) Tasa de pérdida de 0.001%



**Ilustración 34: Media de la ventana de congestión para tcp\_conn\_alt para RTT variable con Loss=0.001%**



**Ilustración 35: Desviación estándar de la ventana de congestión para tcp\_conn\_alt para RTT variable con Loss=0.001%**

### 4.2.1.3 Análisis de para módulo tcp\_conn\_alt con retardo variable

Para las pruebas de ambas pérdidas se aprecia que la media de la ventana de congestión es menor que la media de la referencia, sin embargo, existe una tendencia a que están relacionadas y el mismo comportamiento del módulo lo muestra.

En lo referente a la desviación estándar en todos los casos resultado lo esperado en el sentido que la ventana predicha tiene mucho menos variabilidad que la ventana de congestión controlada por el módulo (esta última no puede ser invariante en el tiempo). El retardo parece no afectar la varianza de las muestras lo cual es interesante dado que gráficamente una prueba con bajo retardo es muy distinta a una con retardo elevado. Por otra parte para las pruebas que utilizaron una pérdida de 0.1% la desviación estándar resultado ser mucho menor que aquellas que usaron pérdida de 0.001%. Esto se verá más en detalle en la siguiente sección de pruebas con tasa de pérdida de paquetes variable.

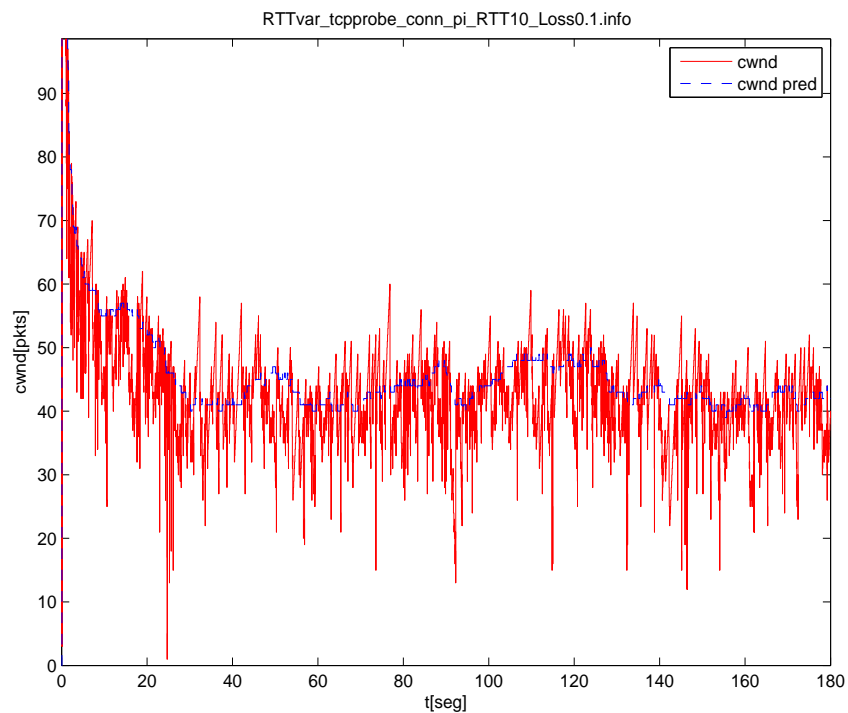
Luego, para comprobar el rendimiento del módulo faltaría comparar tanto la media de éste y la desviación estándar con lo entregado por Reno. Esto se presenta al final de la presente sección.

## 4.2.2 Módulo tcp\_conn\_pi

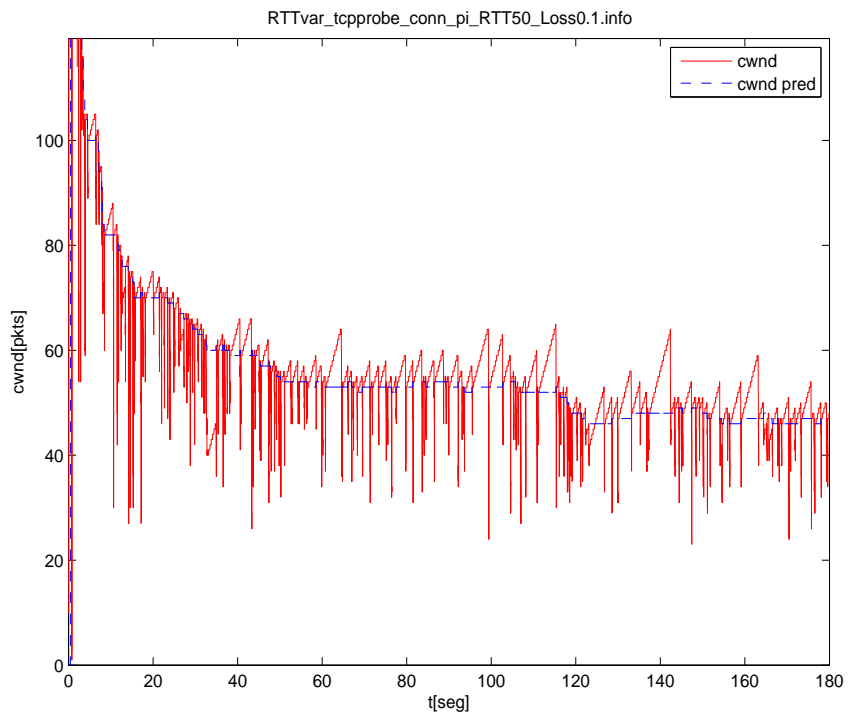
En este caso, dado que se trata de un controlador proporcional integral, se espera que la ventana de congestión esté más cerca de la referencia y que las medias entre la ventana de congestión del módulo y la referencia sean más parecidas que las encontradas en el módulo *tcp\_conn\_alt*.

### 4.2.2.1 Gráficos de ventana de congestión vs tiempo

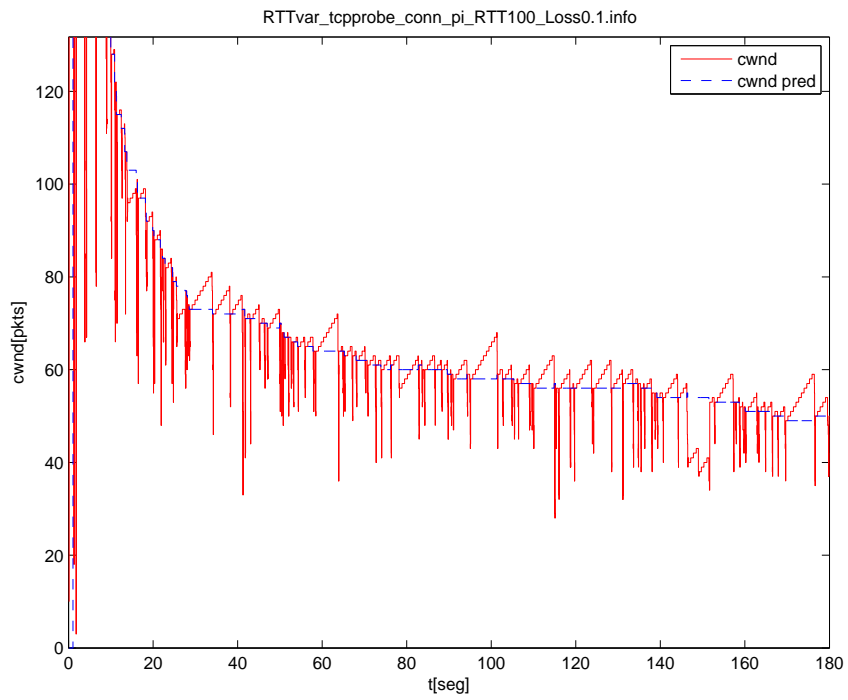
- 1) Tasa de pérdida 0.1%



**Ilustración 36: Módulo tcp\_conn\_pi RTT=40[ms] y Loss=0.1%**

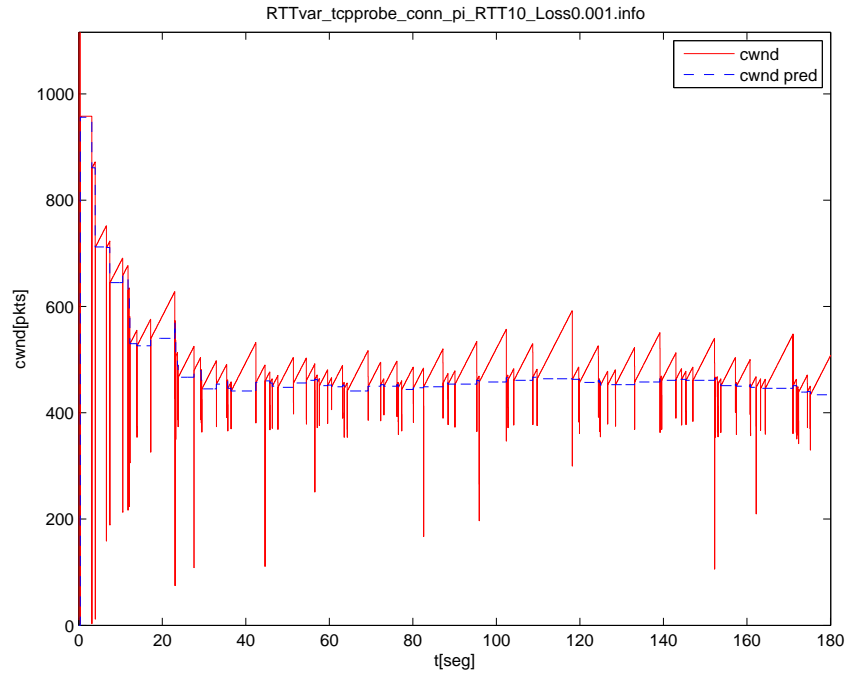


**Ilustración 37: Módulo tcp\_conn\_pi RTT=50[ms] y Loss=0.1%**

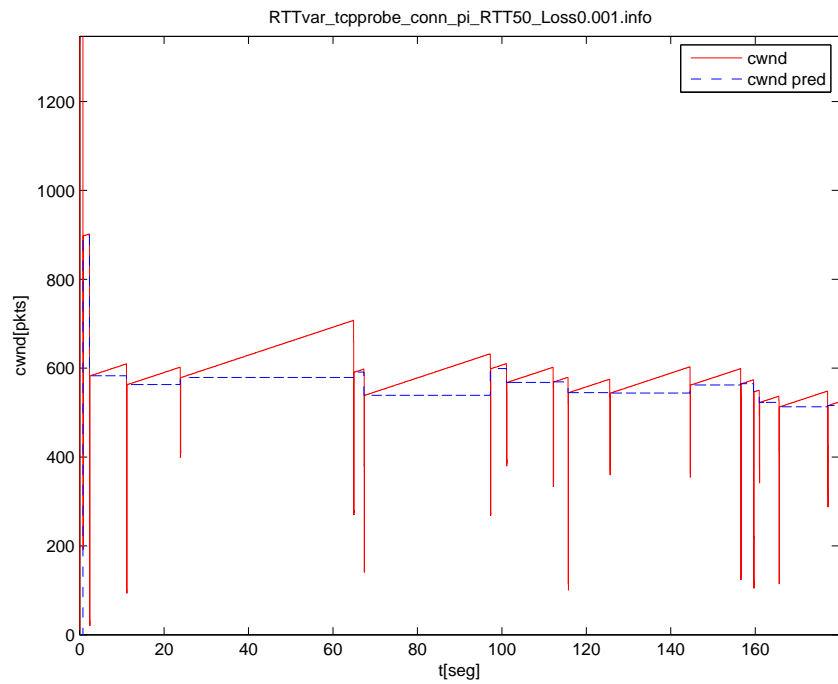


**Ilustración 38: Módulo tcp\_conn\_pi RTT=100[ms] y Loss=0.1%**

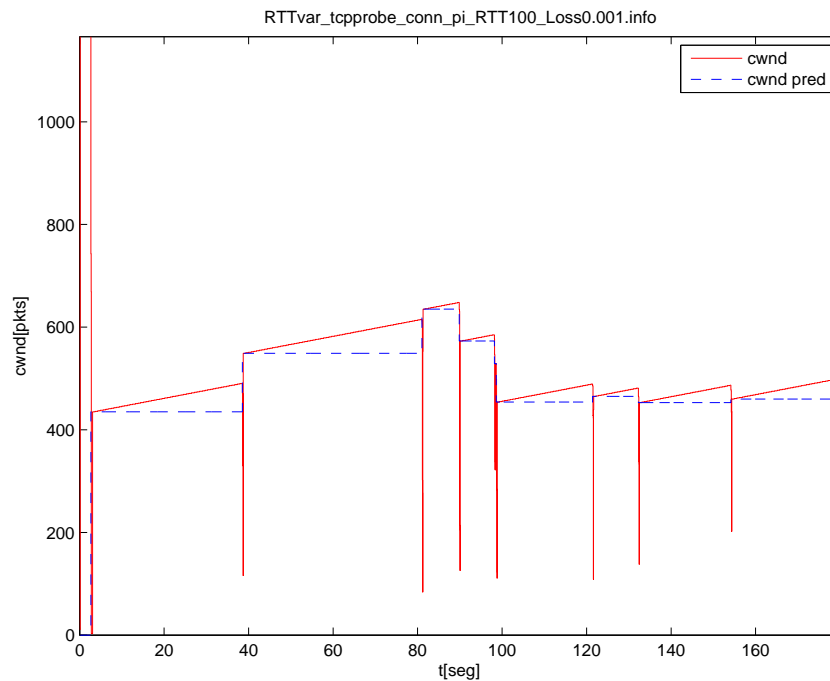
2) Tasa de pérdida 0.001%



**Ilustración 39: Módulo tcp\_conn\_pi RTT=10[ms] y Loss=0.001%**



**Ilustración 40: Módulo tcp\_conn\_pi RTT=50[ms] y Loss=0.001%**



**Ilustración 41: Módulo tcp\_conn\_pi RTT=100[ms] y Loss=0.001%**

El comportamiento de este módulo está acorde a lo previsto. La ventana de congestión sigue más de cerca a la referencia que en el caso anterior y también opuestamente al comportamiento de dicho modulo con un controlador proporcional, en este caso la invariancia temporal de la ventana de congestión hace que esta se mantenga en gran parte sobre la referencia encontrada.

Luego, el primer requisito para considerar el comportamiento del módulo como aceptable se cumple que es seguir la referencia, falta verificar en cuanto a si las medias son parecidas y finalmente comprobar la desviación estándar presente.

### 4.2.2.2 Resultados estadísticos para el módulo tcp\_conn\_alt con RTT variable

1) Tasa de pérdida 0.1%

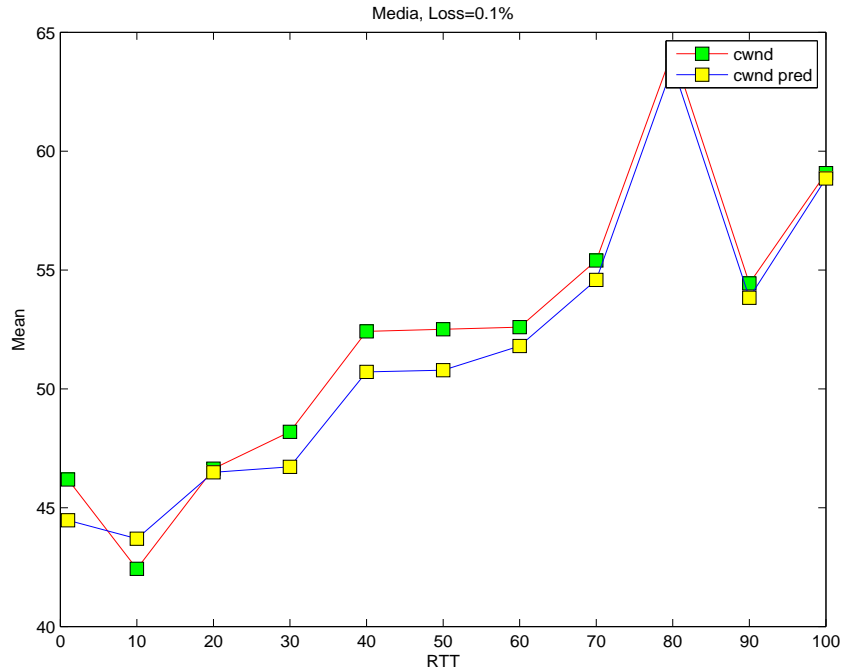


Ilustración 42: Media de la ventana de congestión para tcp\_conn\_pi para RTT variable con Loss=0.1%

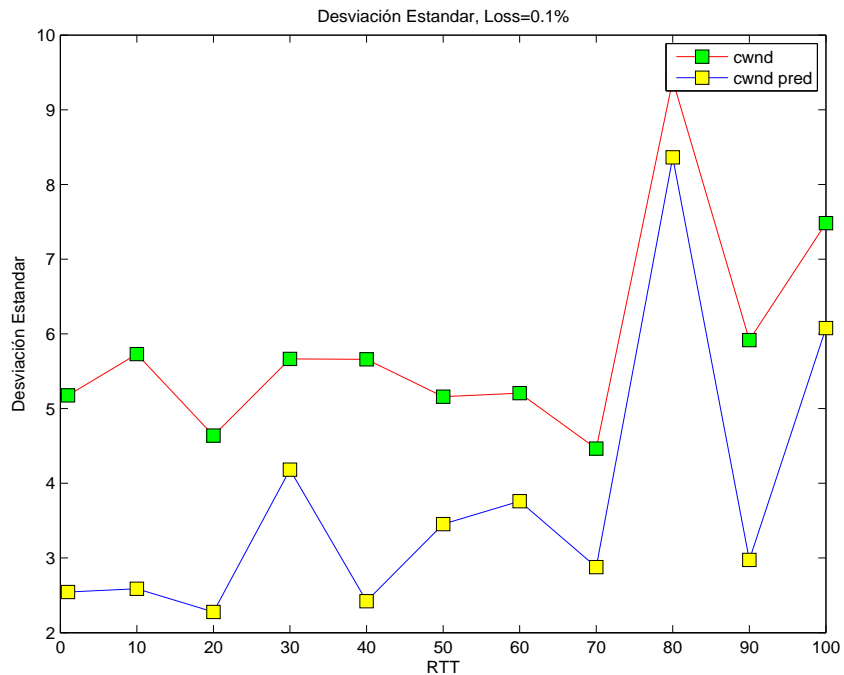


Ilustración 43: Desviación estándar de la ventana de congestión para tcp\_conn\_pi para RTT variable con Loss=0.1%



2) Tasa de pérdida 0.001%

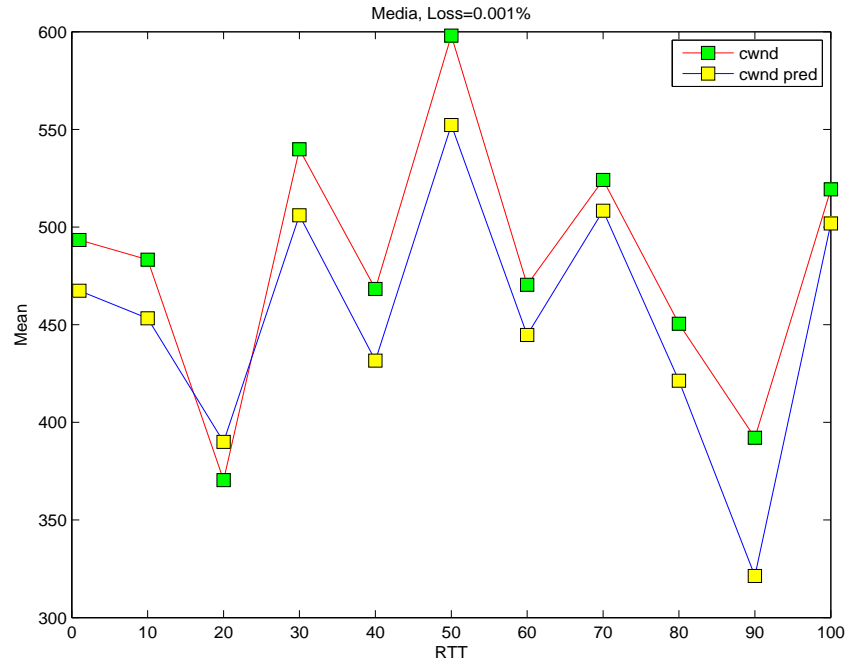


Ilustración 44: Media de la ventana de congestión para tcp\_conn\_pi para RTT variable con Loss=0.001%

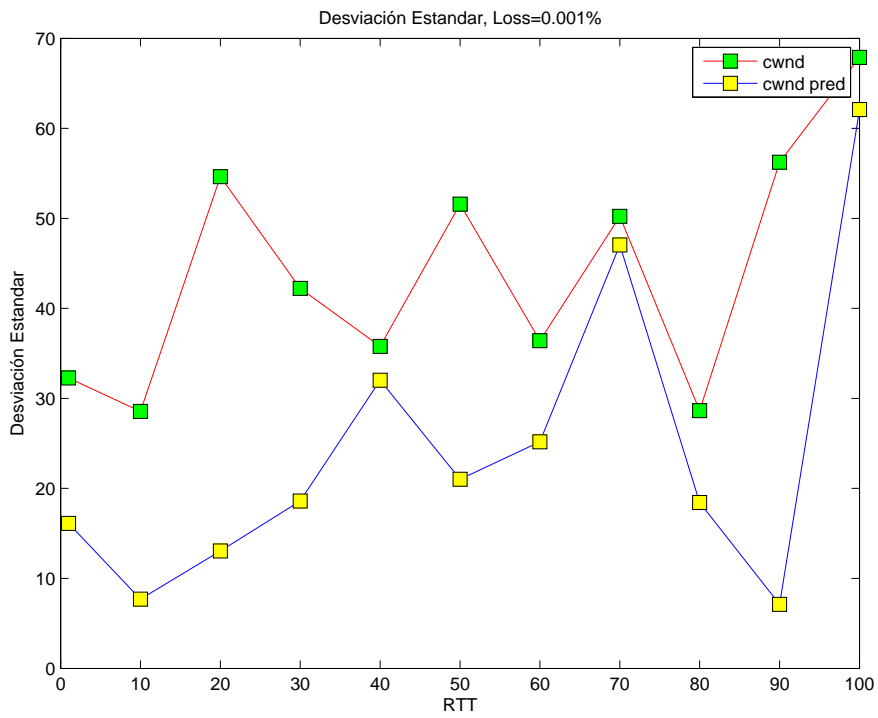


Ilustración 45: Desviación estándar de la ventana de congestión para tcp\_conn\_pi para RTT variable con Loss=0.001%

### 4.2.2.3 Análisis de para módulo *tcp\_conn\_pi* con retardo variable

Este módulo a diferencia de *tcp\_conn\_alt* llega mucho más fácil a la predicción del canal y no se mantiene bajo esta. En ese sentido tiende a ser mucho más estable en cuanto al tamaño de la ventana tendiendo muy pocas veces a estar bajo la referencia.

Las medias de la ventana de congestión obtenidas por el modulo dan resultados alentadores pues tienen una gran semejanza con las medias de la ventana predicha. Sin embargo hay un aspecto que puede ser problemático y corresponde a que en términos de media, la predicción es menor que la ventana de congestión. Esto se produce porque la ventana se encuentra muy pocas veces bajo la predicción y sobre esta está obligada a subir lo más lento posible ( $\alpha \sim 0.1$ ).

En lo referente a la desviación estándar claramente es mayor en la ventana de congestión que en la predicción, sin embargo sus valores son más bajos que los mismos para el módulo *tcp\_conn\_alt*. Por lo tanto este módulo tiene una media más cercana a la referencia y una menor desviación estándar que el otro dando a entender que para efectos generales es más parecido a lo buscado por el proyecto.

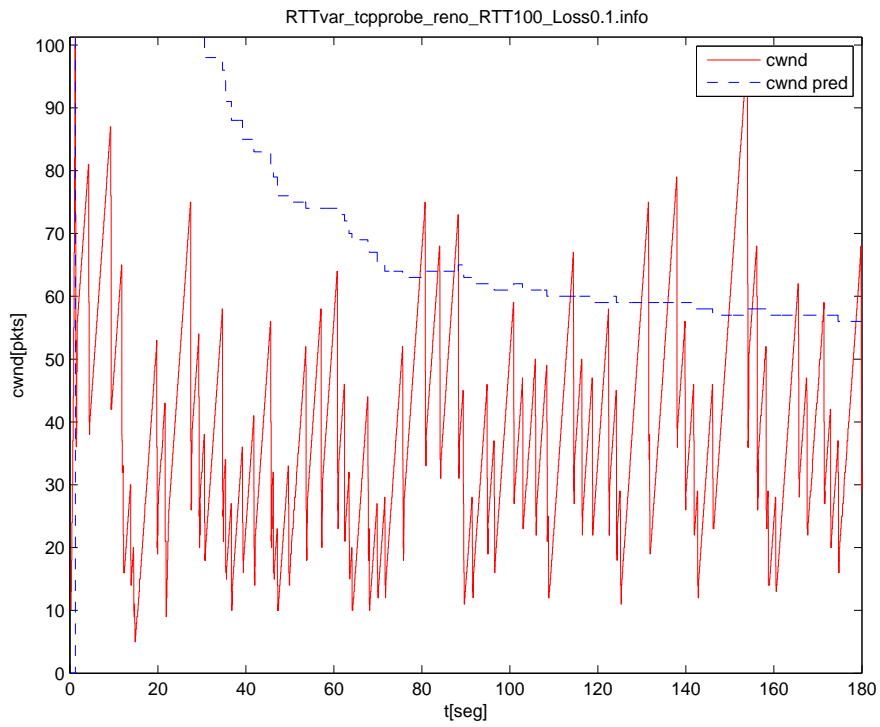
Si la desviación estándar encontrada es comparativamente muy menor a la de Reno, se puede decir que cumple los requisitos.

## 4.2.3 Módulo Reno para análisis comparativo

Con las siguientes pruebas del módulo Reno únicamente se busca mostrar la dispersividad y el comportamiento de *probing* de dicho modulo y así tener valores para comparar con los módulos propios del proyecto.

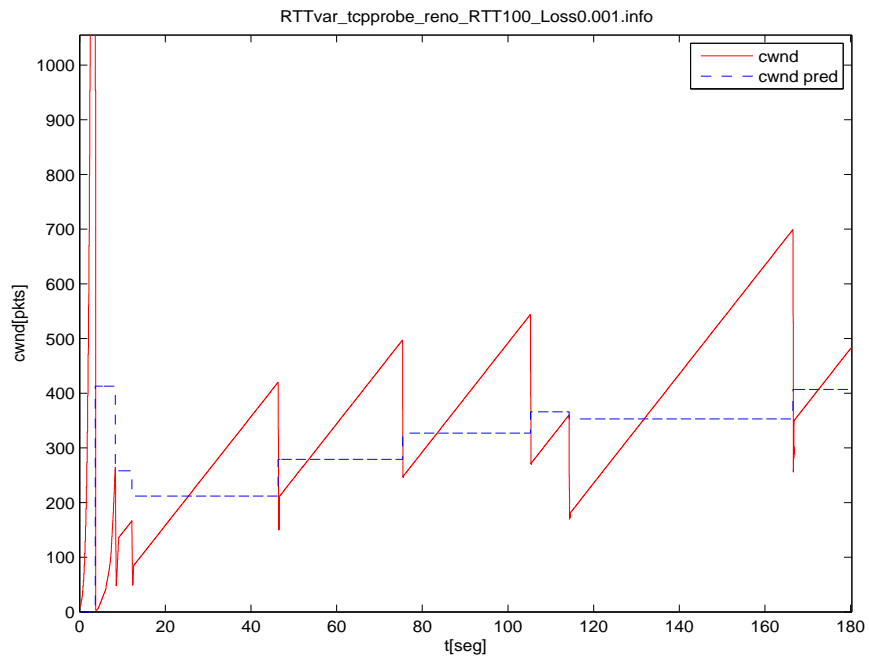
### 4.2.3.1 Gráficos de ventana de congestión vs tiempo

1) Tasa de pérdida 0.1%



**Ilustración 46: Módulo tcp\_reno RTT=100[ms] y Loss=0.1%**

2) Tasa de pérdida 0.001%



**Ilustración 47: Módulo tcp\_reno RTT=100[ms] y Loss=0.001%**

### 4.2.3.2 Resultados estadísticos para el módulo tcp\_reno con RTT variable

1) Tasa de pérdida 0.1%

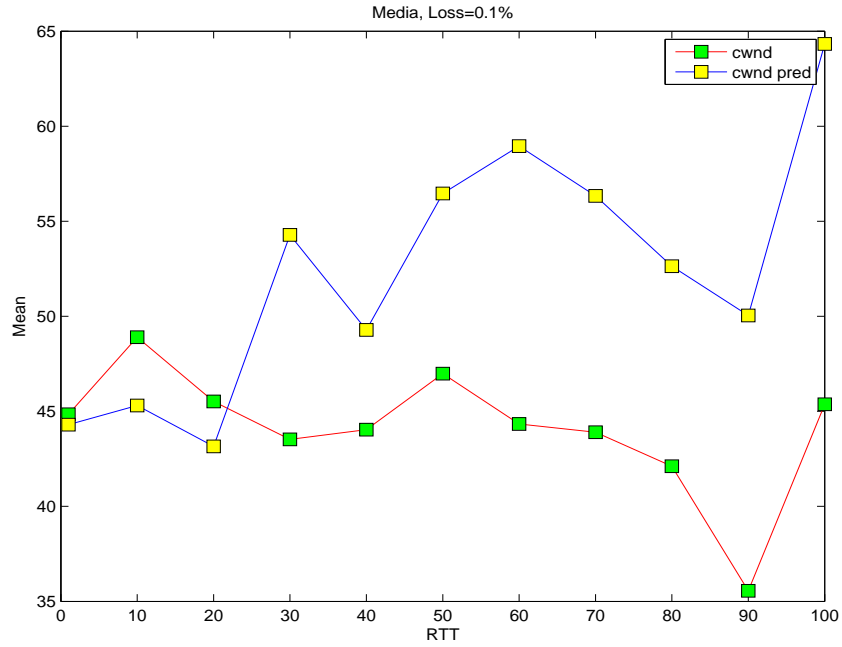


Ilustración 48: Media de la ventana de congestión para tcp\_reno para RTT variable con Loss=0.1%

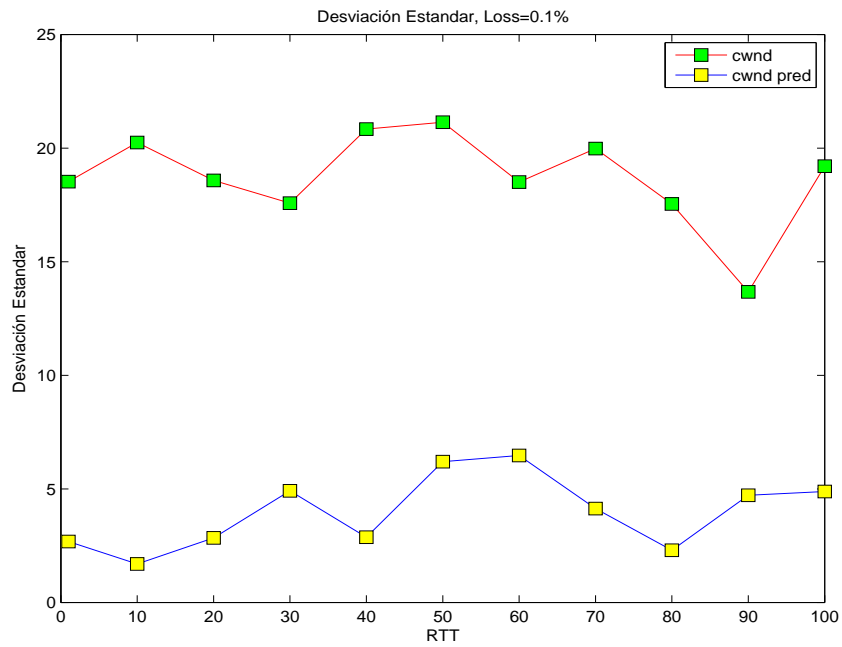
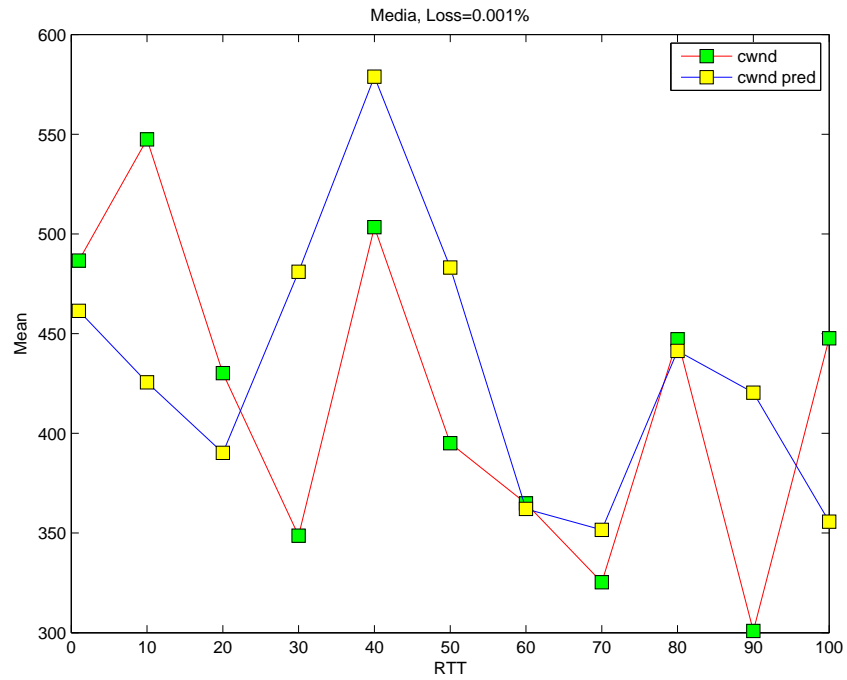
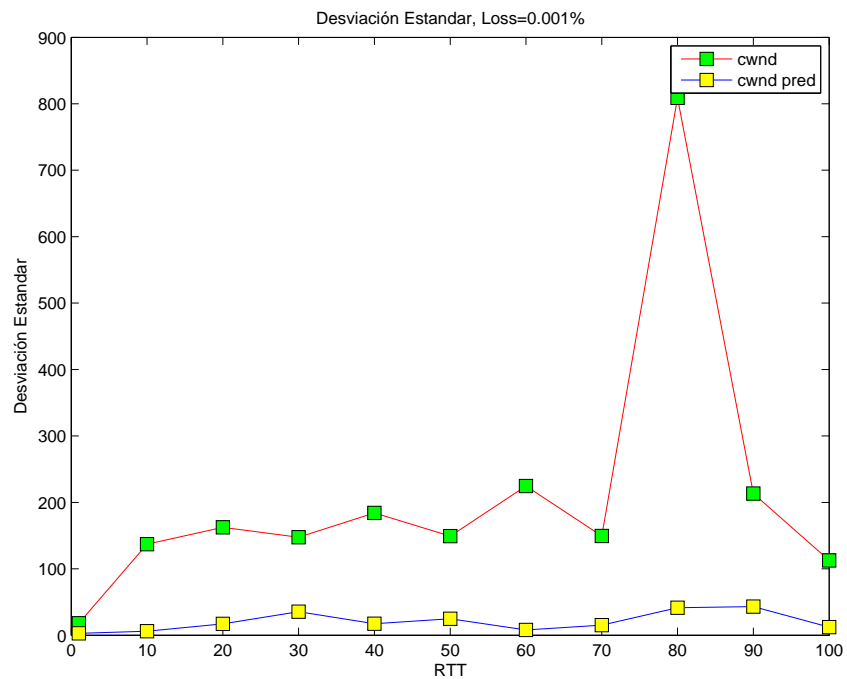


Ilustración 49: Desviación estándar de la ventana de congestión para tcp\_reno para RTT variable con Loss=0.1%

2) Tasa de pérdida 0.001%



**Ilustración 50: Media de la ventana de congestión para tcp\_reno para RTT variable con Loss=0.001%**



**Ilustración 51: Media de la ventana de congestión para tcp\_reno para RTT variable con Loss=0.001%**

### 4.2.3.3 Análisis de para módulo tcp\_reno con retardo variable

Como se muestra tanto en los gráficos de la ventana de congestión vs tiempo así como en los gráficos de la media y desviación estándar el algoritmo Reno, comparado a los módulos implementados del proyecto, es mucho más “ruidoso” dado su funcionamiento mediante *probing* donde se aprecia claramente por su alta desviación estándar, sin embargo las medias de ventana de congestión y ventana predicha tiene un parecido significativo lo cual es lógico pues la predicción se obtuvo de la fórmula de *throughput* para Reno.

Con esto queda bastante comprobado que los módulos del proyecto están bien planteados: Siguen una referencia, a pesar de no converger por razones técnicas la varianza presente es mucho menor que utilizar Reno, las medias de los tres módulos son cercanas y las oscilaciones presentes son estables alrededor de la predicción de la capacidad.

## 4.3 Resultados de tasa de pérdida variable

A diferencia de las pruebas anteriores en donde los resultados de la ventana de congestión en el tiempo para distintos *RTT* y para cada módulo utilizado tenían solo un valor de estudio al comparar estadísticamente *cwnd* con *cwnd\_pred* para simulaciones con el mismo *RTT*. En cambio en esta prueba, dado que efectivamente la pérdida si influye en el tamaño de la ventana de congestión, si importa como varia esta para los distintos valores aplicados así como los resultados estadísticos obtenidos. También la pérdida es de suma importancia pues define el progreso de la actualización de la ventana de congestión predicha, dado que se actualiza mediante  $\Delta$  y para ello se requiere la presencia de pérdidas. Si estas son muy infrecuentes la predicción tomara un tiempo en llegar hasta el valor correcto.

Las siguientes pruebas tienen como valores utilizados de pérdida de la red de 0.1%, 0.01% y 0.001%, se utiliza un peso de  $\frac{1}{n}$  con  $n$  la cantidad de iteraciones para el cálculo de la referencia. Se utilizaron dos valores distintos de *RTT*: 10[ms] y 100[ms].

Una prueba que analiza el comportamiento del módulo para distintos estados de pérdida de la red es importante estudiarla porque la tasa de pérdida sirve como indicador de congestión de la red. Por lo tanto estas pruebas tienen como propósito mostrar el funcionamiento de los modulo para condiciones de poca congestión a elevada congestión.

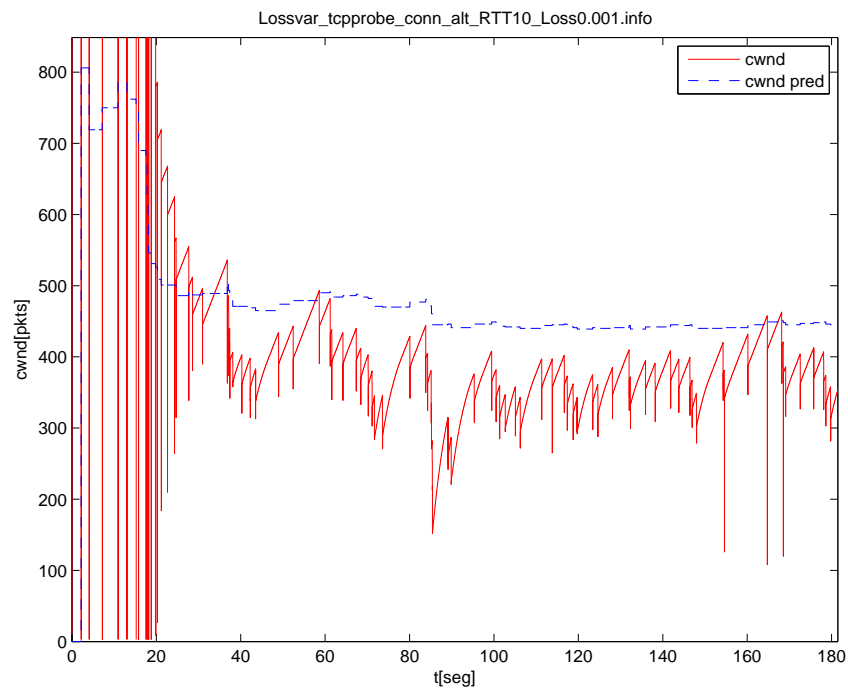
En este caso también se espera que el comportamiento de ambos módulos muestre el seguimiento de la ventana de congestión a la ventana predicha independiente de la tasa de pérdida presente en el canal. Igualmente para que un resultado sea considerado bueno se requiere también que la desviación estándar de la ventana de congestión no sea elevada tomado la del módulo Reno como referencia comparativa para todas las pérdidas utilizadas, esto dado que los módulos por diseño y limitaciones van a tener un comportamiento oscilatorio en donde se espera que sea lo más reducido posible.

### 4.3.1 Módulo tcp\_conn\_alt

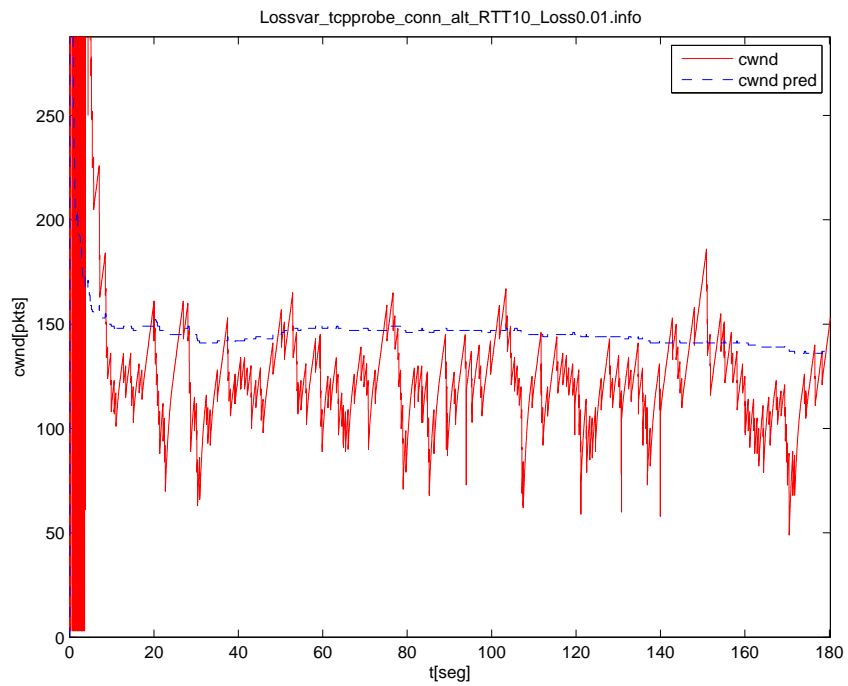
En este caso se busca ver el comportamiento de este módulo que utiliza un controlador proporcional porcentual respecto a las distintas tasas de pérdida aplicadas al canal. Se espera que independiente de la pérdida aplicada el algoritmo tenga el comportamiento deseado.

#### 4.3.1.1 Gráficos de ventana de congestión vs tiempo

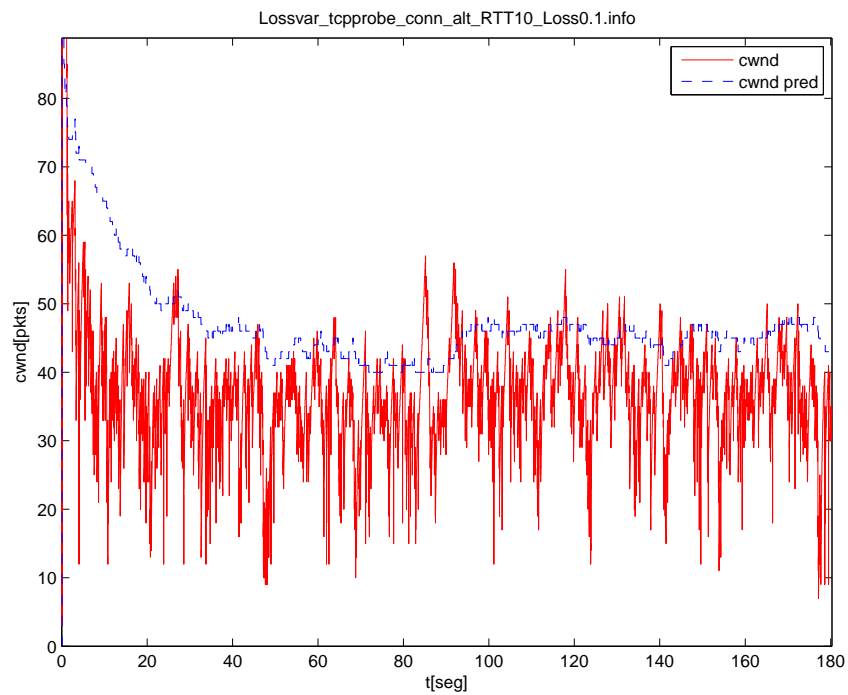
1) *RTT* de 10[ms]



**Ilustración 52: Módulo tcp\_conn\_alt RTT=10[ms] y Loss=0.001%**



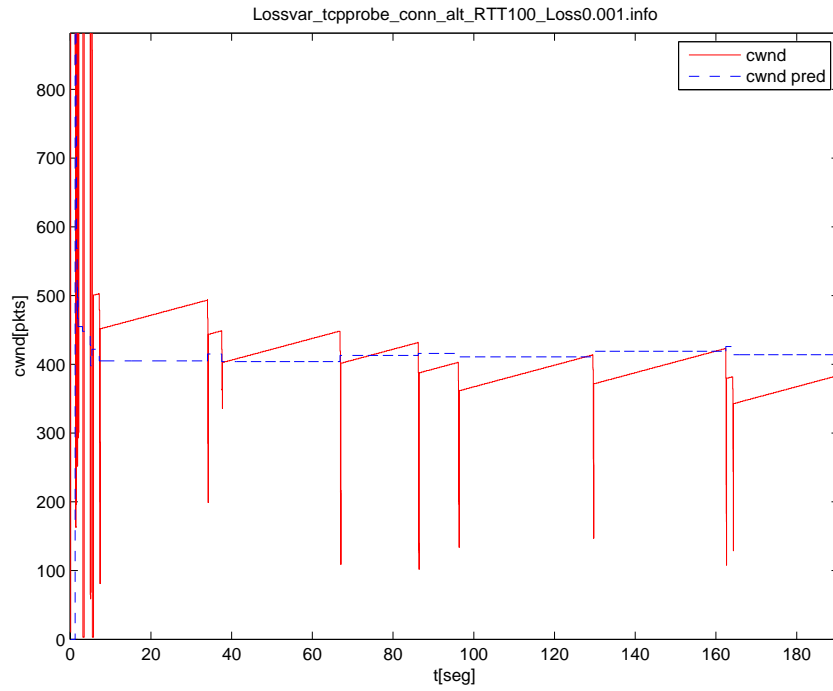
**Ilustración 53: Módulo tcp\_conn\_alt RTT=10[ms] y Loss=0.01%**



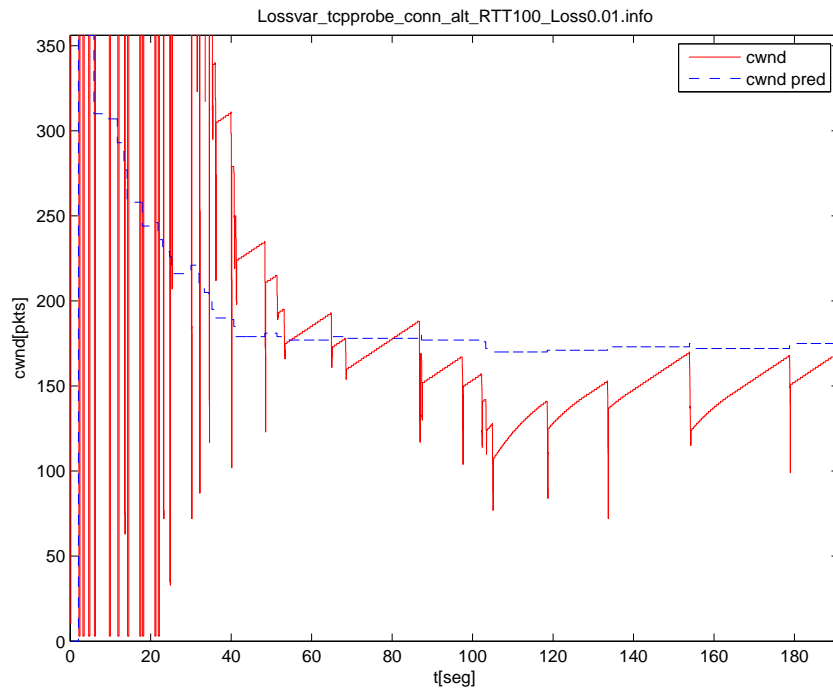
**Ilustración 54: Módulo tcp\_conn\_alt RTT=10[ms] y Loss=0.1%**



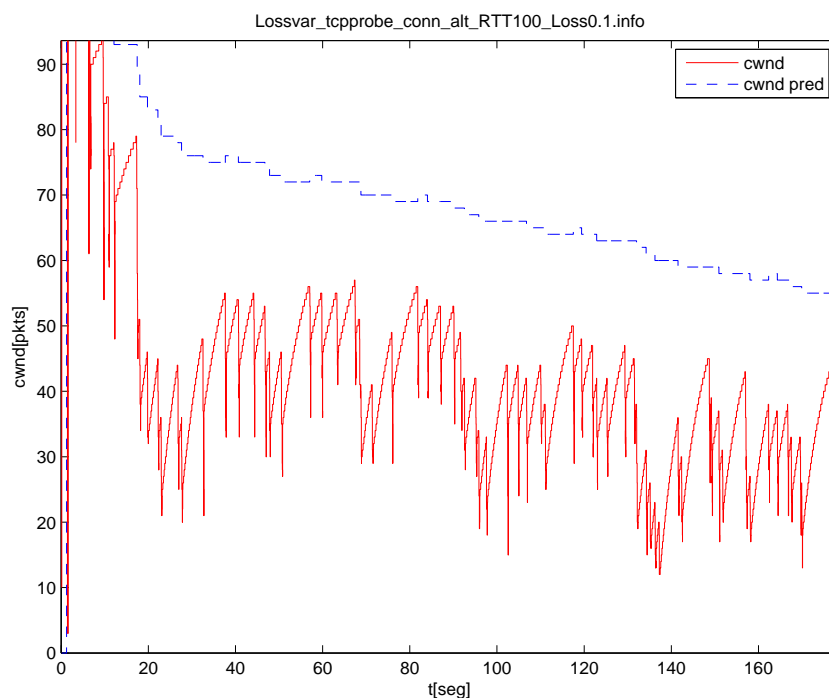
2) *RTT* de 100[ms]



**Ilustración 55: Módulo tcp\_conn\_alt RTT=100[ms] y Loss=0.001%**



**Ilustración 56: Módulo tcp\_conn\_alt RTT=100[ms] y Loss=0.01%**



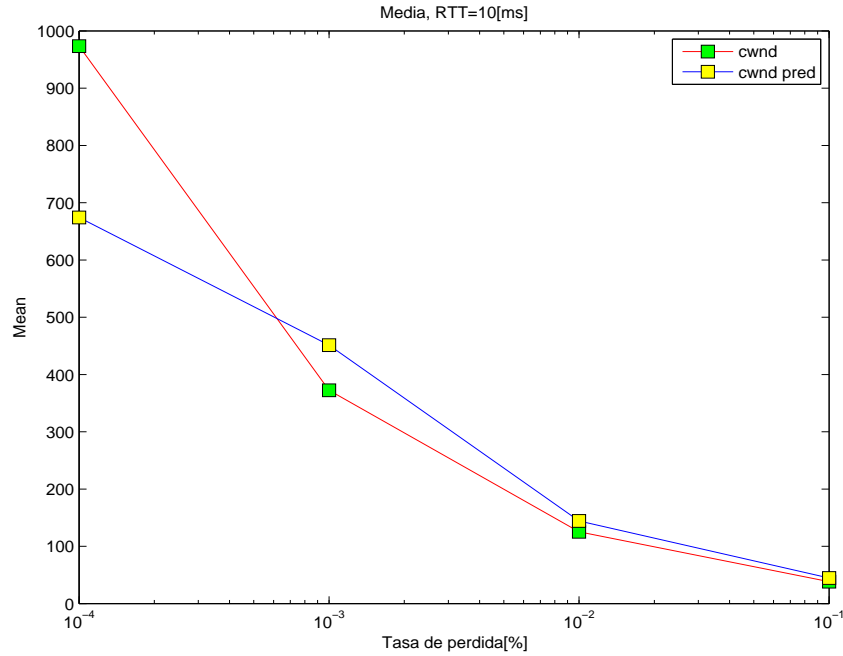
**Ilustración 57: Módulo tcp\_conn\_alt RTT=100[ms] y Loss=0.1%**

Este módulo tiene un comportamiento esperado para todos los casos de  $RTT=10[ms]$  en el sentido de seguir la referencia. Para los casos utilizando un  $RTT$  de  $100[ms]$  cuando la tasa de pérdida es alta de  $0.1\%$  el modulo tiene problemas para mantenerse cerca de la referencia encontrada. Prácticamente también se repite lo visto en las pruebas con  $RTT$  variable en donde el comportamiento de este protocolo mantiene la ventana de congestión gran parte del tiempo bajo la ventana predicha lo cual está dentro del comportamiento esperado. Los mejores resultados ocurren a un retardo elevado de  $100[ms]$  y pérdidas bajas ( $0.001\%$  y  $0.01\%$ ) dado que son los casos en donde más cerca se encuentra la ventana de congestión de la ventana predicha.

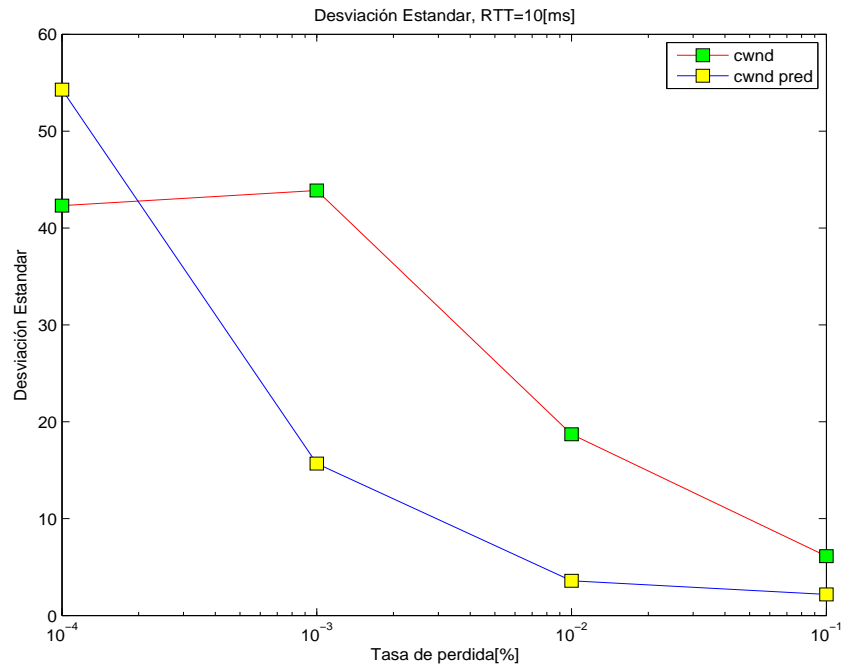
Adicionalmente, si la desviación estándar encontrada en estas pruebas resulta ser menor que en el caso de Reno el modulo está funcionando de acuerdo a lo esperado.

### 4.3.1.2 Resultados estadísticos para el módulo tcp\_conn\_alt con tasa de pérdida variable

1) *RTT* de 10[ms]

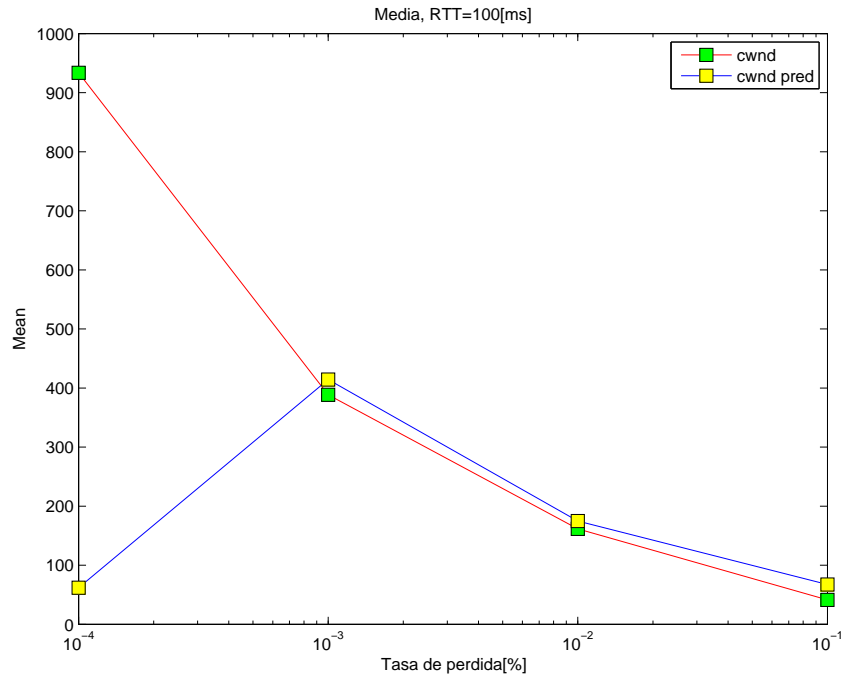


**Ilustración 58:** Media de la ventana de congestión para tcp\_conn\_alt para tasa de pérdida variable con *RTT*=10[ms]

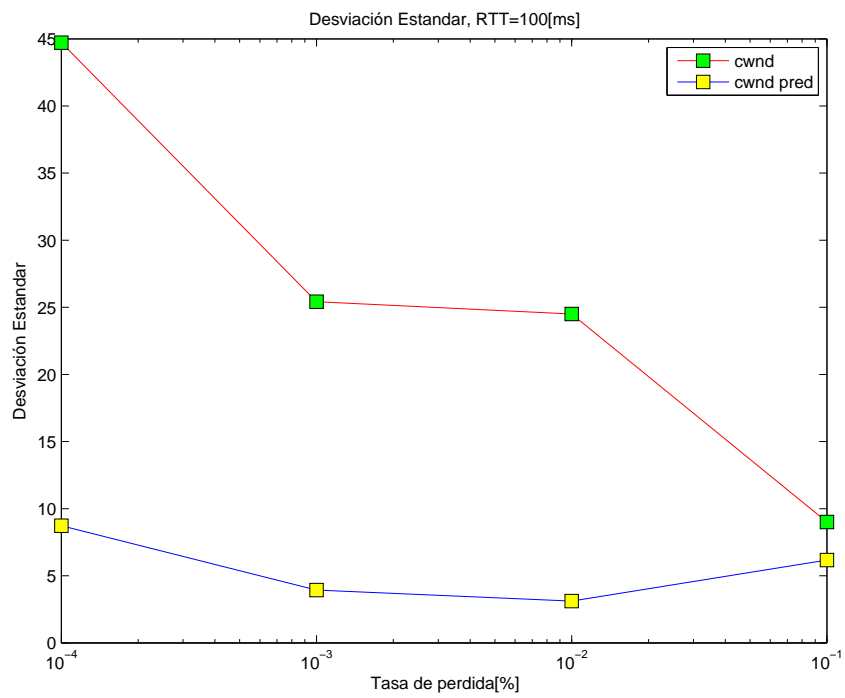


**Ilustración 59:** Desviación estándar de la ventana de congestión para tcp\_conn\_alt para tasa de pérdida variable con *RTT*=10[ms]

2)  $RTT$  de 100[ms]



**Ilustración 60: Media de la ventana de congestión para tcp\_conn\_alt para tasa de pérdida variable con  $RTT=100$ [ms]**



**Ilustración 61: Desviación estándar de la ventana de congestión para tcp\_conn\_alt para tasa de pérdida variable con  $RTT=100$ [ms]**

### 4.3.1.3 Análisis de para módulo tcp\_conn\_alt con tasa de pérdida variable

Al igual que con la prueba de retardo variable el modulo presenta comportamiento oscilatorio de incremento lineal y decremento multiplicativo parecido a *TCP-Reno* pero de amplitudes mucho menores, y también la oscilación ocurre bajo la predicción.

Las medias entre la ventana de congestión y la ventana predicha están bastante parecidas tanto para  $RTT=10[ms]$  como para  $RTT=100[ms]$  lo cual es un buen indicador del comportamiento del módulo. La media de la ventana de congestión está bajo la predicción en todos los casos salvo para  $RTT=10[ms]$  y tasa de pérdida de 0.0001% que se debe principalmente a que el valor de la ventana predicha es impreciso por la falta de muestras de  $\Delta$  dada la baja frecuencia de pérdidas.

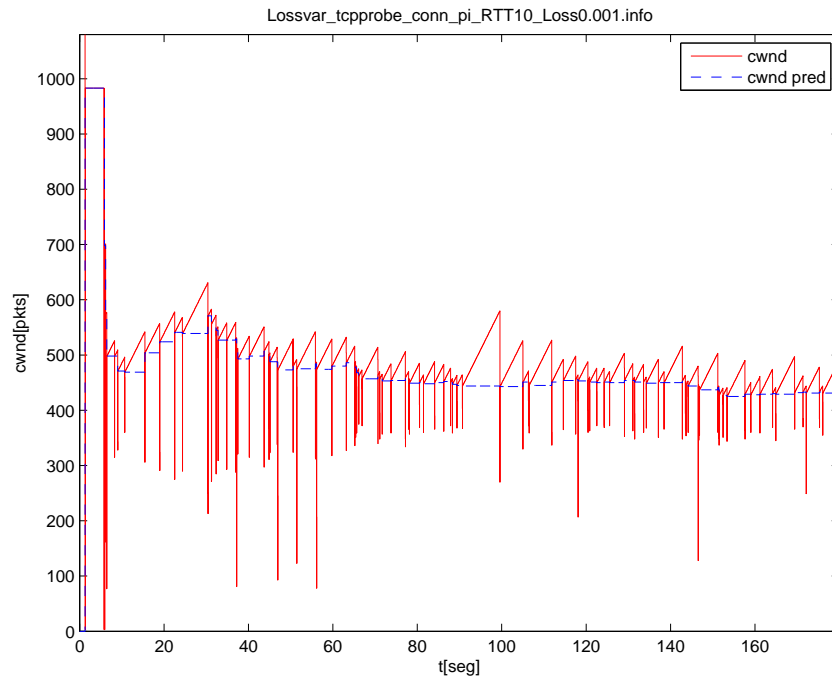
En esta prueba la desviación estándar tiende a reducirse a medida que la tasa de pérdida aumenta, lo cual es un comportamiento esperado dado que sobre la ventana predicha la ventana de congestión está obligada a subir gradualmente hasta presentarse una pérdida, por lo tanto a menor pérdida la ventana tiene más probabilidades de subir. Además la tasa de pérdida tiene una relación directa con el tamaño de la ventana de congestión produciéndose amplitudes mayores a menor pérdida dada la fórmula de *throughput* de reno:  $Throughput \sim \frac{1}{\sqrt{p}}$ . Esto resulta curioso en una primera instancia dado que un canal robusto con una tasa de pérdidas bajísima trabajando con este módulo de evasión de congestión tendería a tener una varianza mayor.

### 4.3.2 Módulo tcp\_conn\_pi

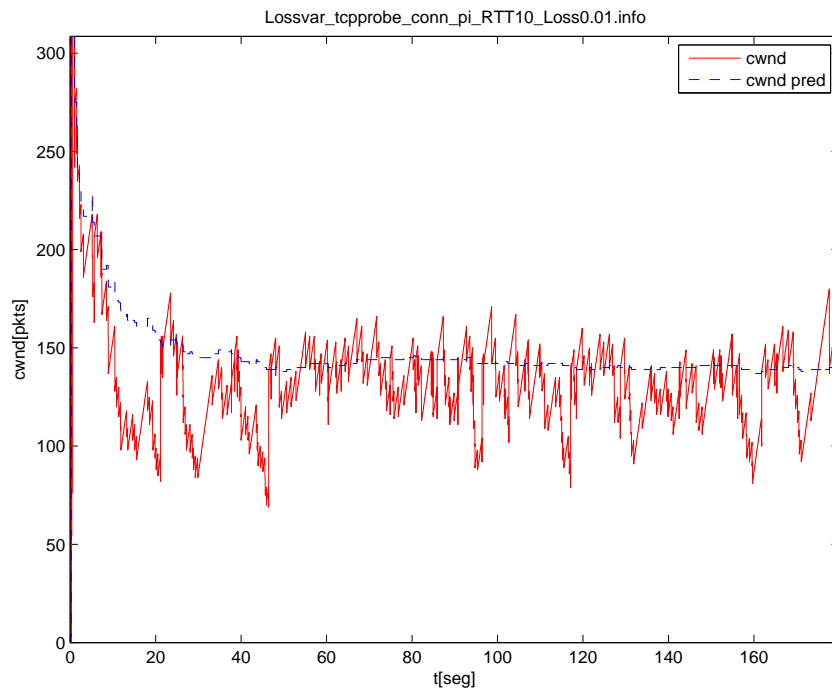
En este caso se busca encontrar como reacción este módulo con un controlador proporcional integral a las distintas tasas de pérdida.

### 4.3.2.1 Gráficos de ventana de congestión vs tiempo

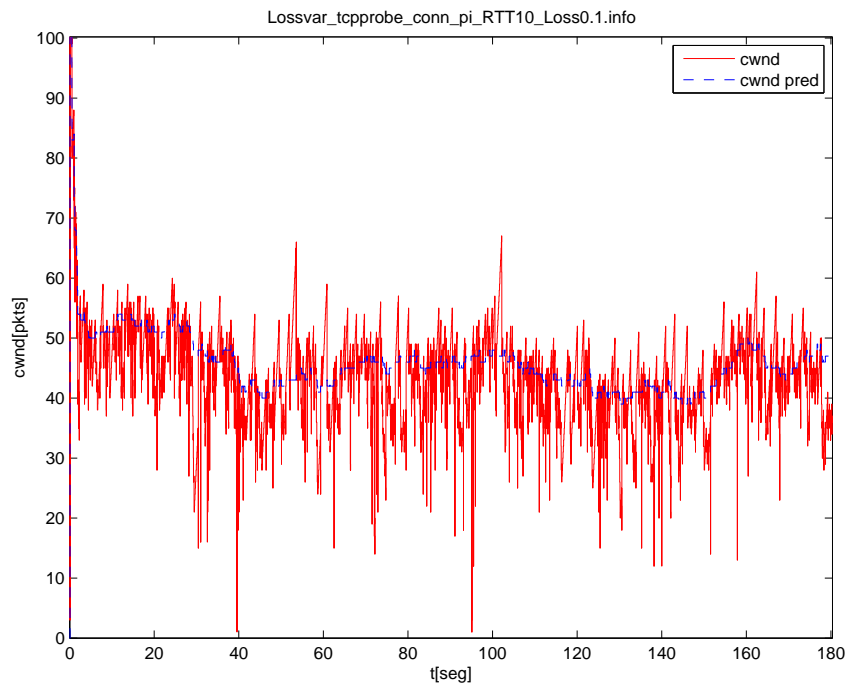
1) *RTT* de 10[ms]



**Ilustración 62: Módulo tcp\_conn\_pi RTT=10[ms] y Loss=0.001%**

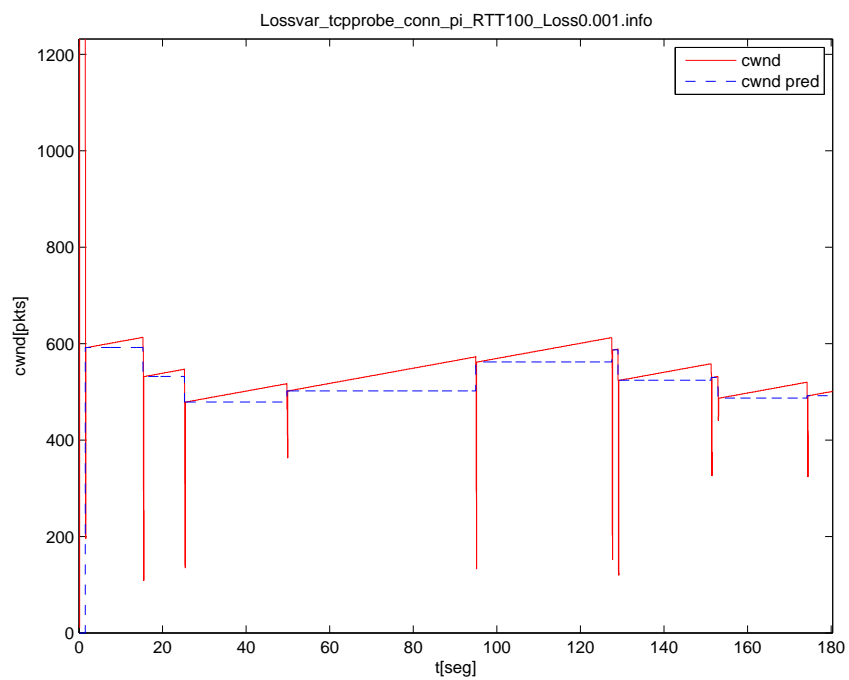


**Ilustración 63: Módulo tcp\_conn\_pi RTT=10[ms] y Loss=0.01%**

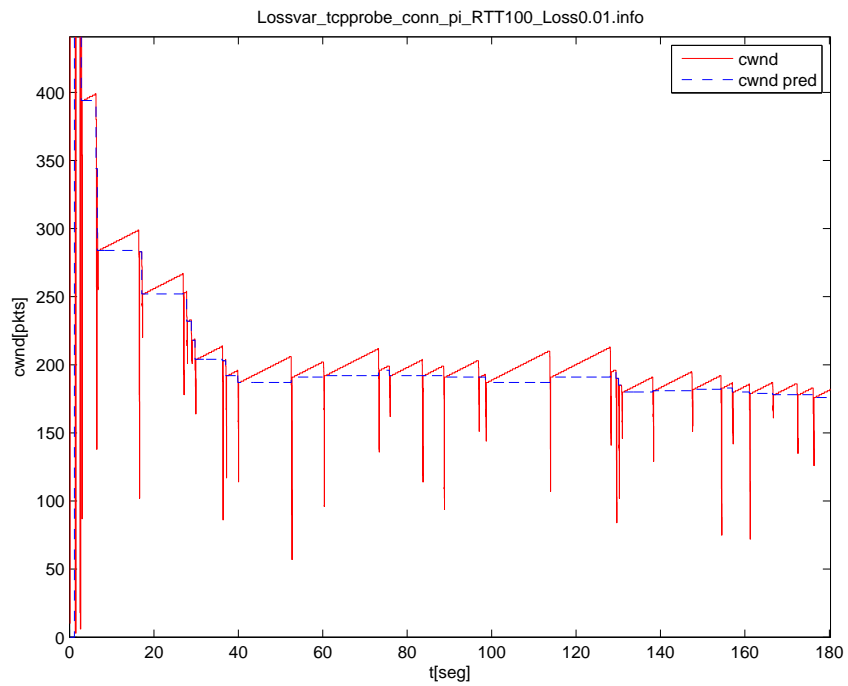


**Ilustración 64: Módulo tcp\_conn\_pi RTT=10[ms] y Loss=0.1%**

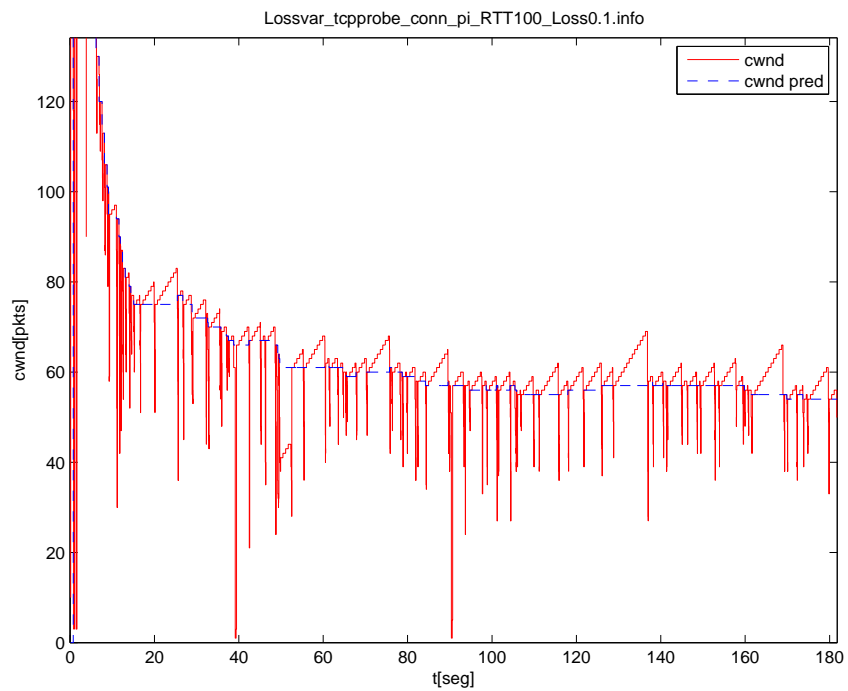
2) *RTT* de 100[ms]



**Ilustración 65: Módulo tcp\_conn\_pi RTT=100[ms] y Loss=0.001%**



**Ilustración 66: Módulo tcp\_conn\_pi RTT=100[ms] y Loss=0.01%**



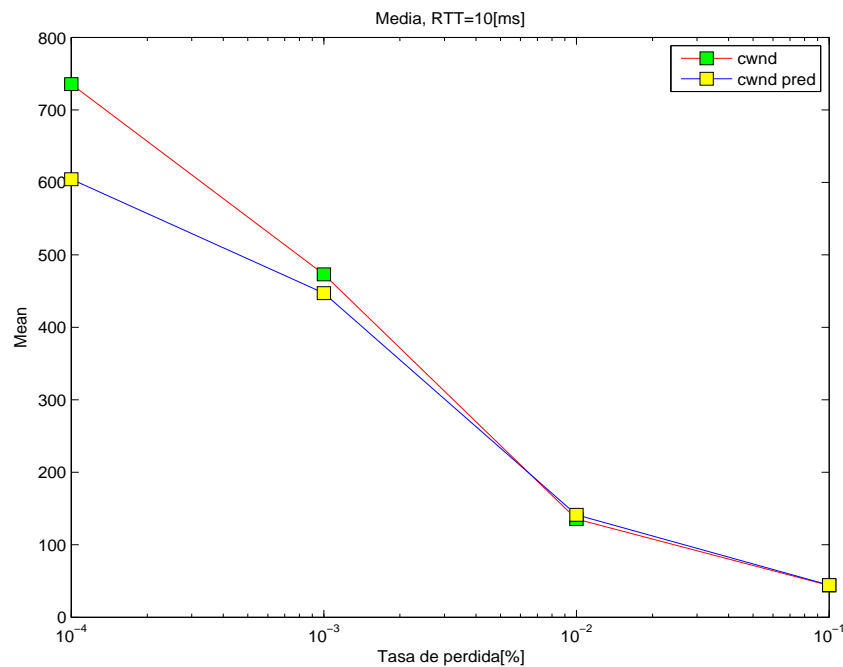
**Ilustración 67: Módulo tcp\_conn\_pi RTT=100[ms] y Loss=0.1%**



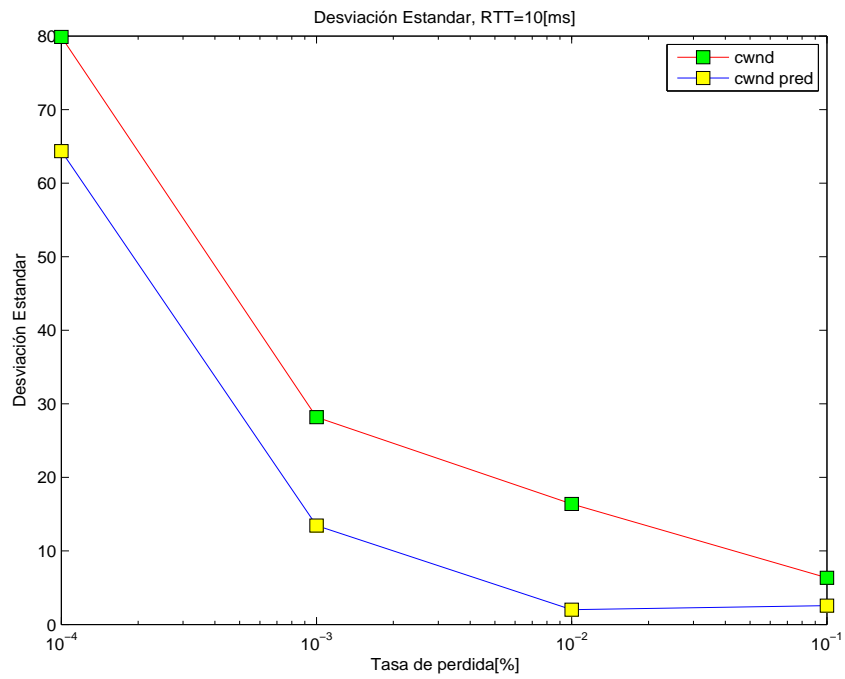
Este módulo prácticamente alcanza la referencia en todos los casos independiente de la pérdida aplicada a diferencia del módulo *tcp\_conn\_alt* en donde para un retardo y una pérdida elevados presenta un comportamiento no adecuado. A priori y en base a los resultados previos la ventana se mantiene cerca de la referencia y sobre ésta. En los casos de *RTT=10[ms]* dado la rápida respuesta de los paquetes transmitidos se observa un comportamiento ruidoso y más disperso a medida que la pérdida se eleva, sin embargo mientras comparativamente la varianza presente sea menor que la de Reno, el comportamiento del módulo esta correcto y dentro de los márgenes esperados.

#### 4.3.2.2 Resultados estadísticos para el módulo *tcp\_conn\_pi* con tasa de pérdida variable

1) *RTT* de 10[ms]

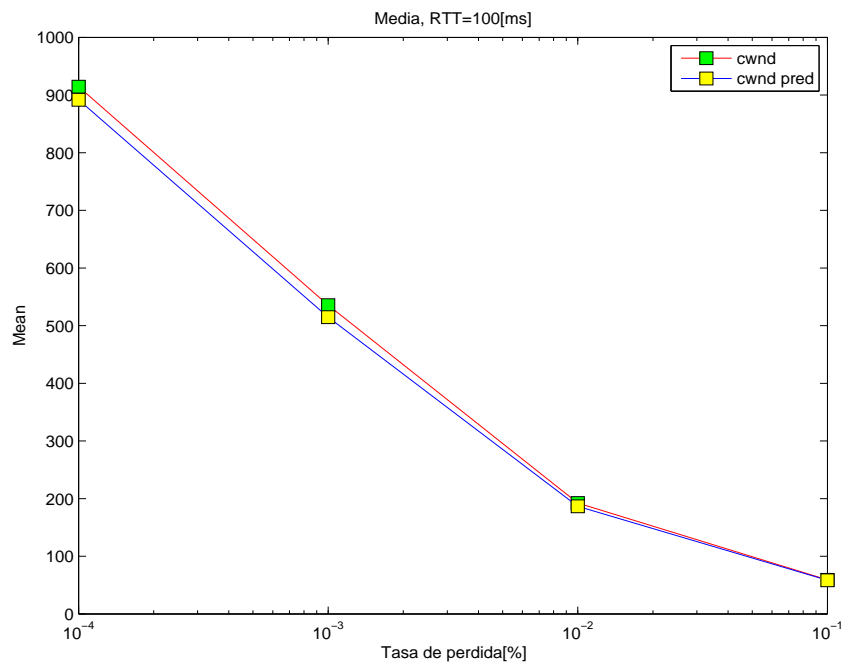


**Ilustración 68: Media de la ventana de congestión para *tcp\_conn\_pi* para tasa de pérdida variable con *RTT=10[ms]***

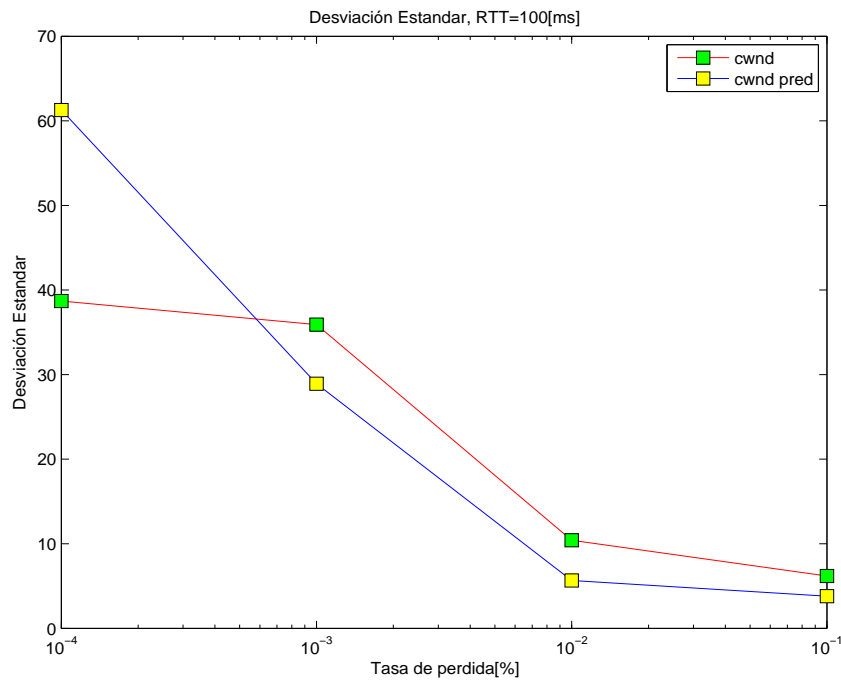


**Ilustración 69: Desviación estándar de la ventana de congestión para tcp\_conn\_pi para tasa de pérdida variable con RTT=10[ms]**

2) *RTT* de 100[ms]



**Ilustración 70: Media de la ventana de congestión para tcp\_conn\_pi para tasa de pérdida variable con RTT=100[ms]**



**Ilustración 71: Desviación estándar de la ventana de congestión para tcp\_conn\_pi para tasa de pérdida variable con RTT=100[ms]**

#### 4.3.2.3 Análisis de para módulo tcp\_conn\_pi con tasa de pérdida variable

La media de la ventana de congestión de este módulo es prácticamente la media de la ventana predicha lo cual es un buen indicador comparándose con el módulo *tcp\_conn\_alt*. Por otra parte las desviaciones estándar son parecidas independientemente del módulo a utilizar de manera que la media corresponderá al factor de decisión para evaluar al módulo más acorde a los fines del proyecto.

Se repite la misma tendencia apreciable en el otro modulo en donde la desviación estándar aumenta acorde disminuye la tasa de pérdidas del canal. Las causas de este fenómeno son las mismas: a menor pérdida la ventana de congestión es mayor produciéndose oscilaciones mayores y también a menor pérdida el incremento aditivo sobre la ventana predicha es probabilísticamente más largo.

### 4.3.3 Módulo Reno para análisis comparativo

Con las siguientes pruebas del módulo Reno únicamente se busca mostrar la dispersividad y el comportamiento de *probing* de dicho modulo y así tener valores para comparar con los módulos propios del proyecto.

#### 4.3.3.1 Gráficos de ventana de congestión vs tiempo

1) *RTT* de 10[ms]

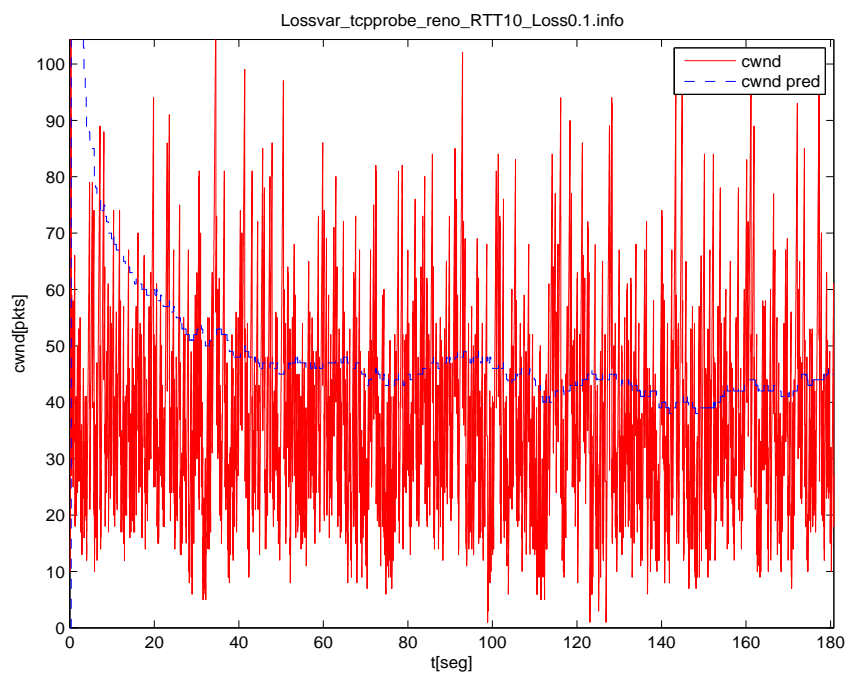


Ilustración 72: Módulo tcp\_reno RTT=10[ms] y Loss=0.1%

2)  $RTT$  de 100[ms]

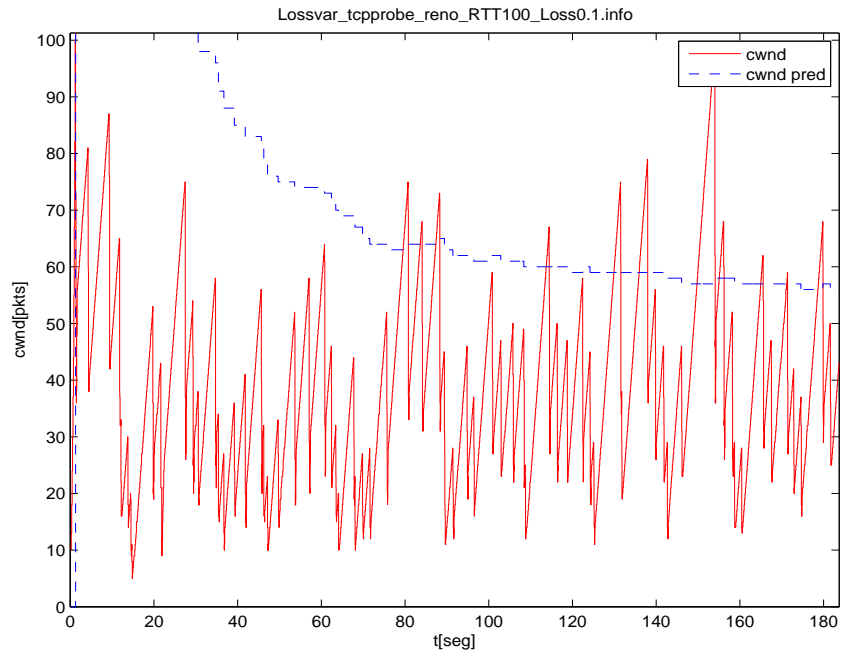


Ilustración 73: Módulo tcp\_reno  $RTT=100$ [ms] y  $Loss=0.1\%$

### 4.3.3.2 Resultados estadísticos para el módulo tcp\_reno con tasa de pérdida variable

1)  $RTT=10$ [ms]

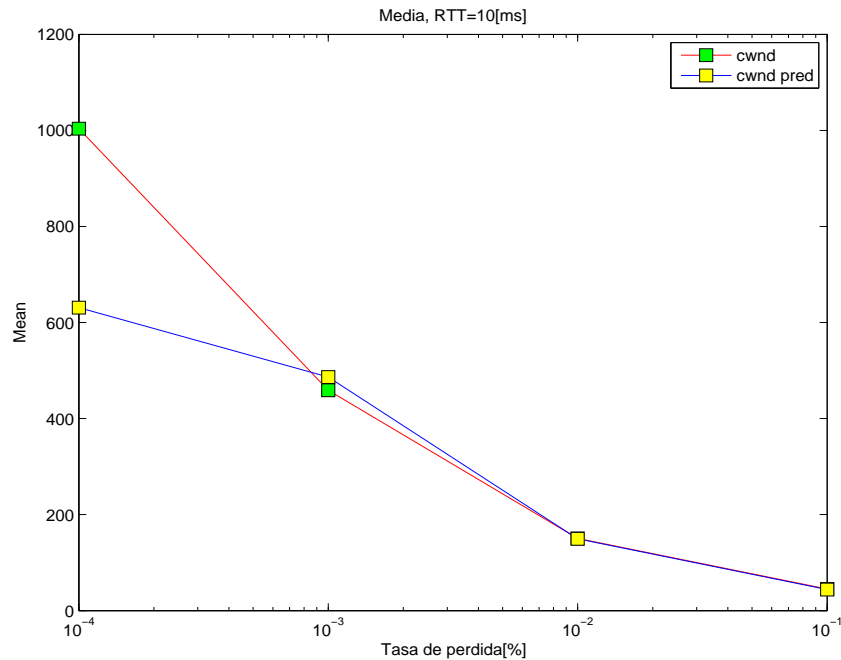
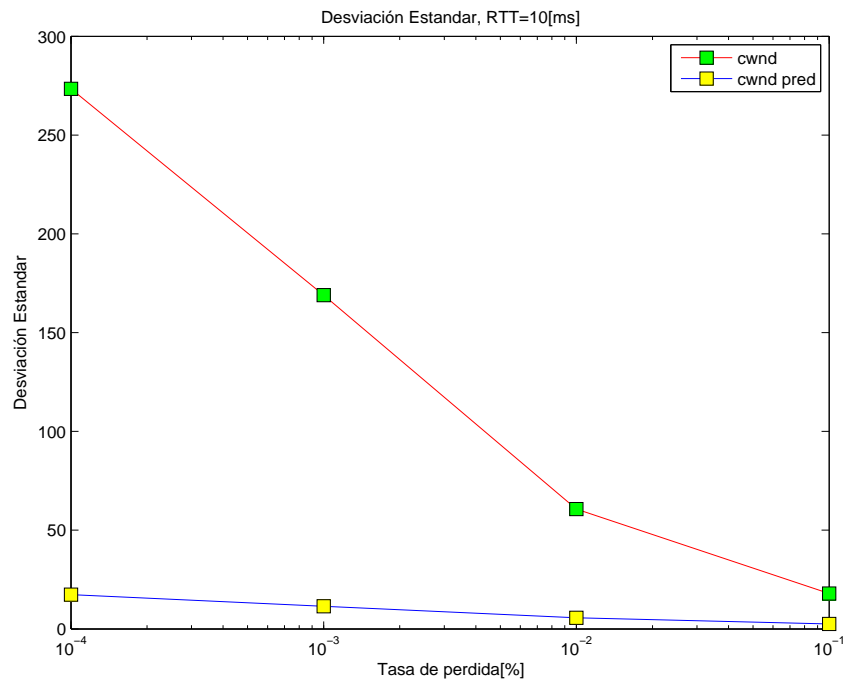
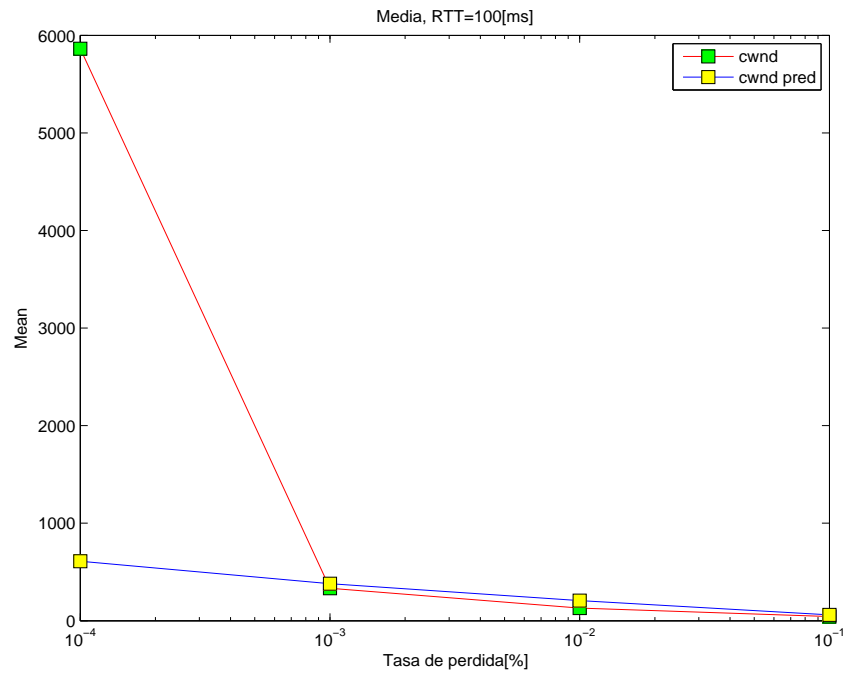


Ilustración 74: Media de la ventana de congestión para tcp\_reno para tasa de pérdida variable con  $RTT=10$ [ms]

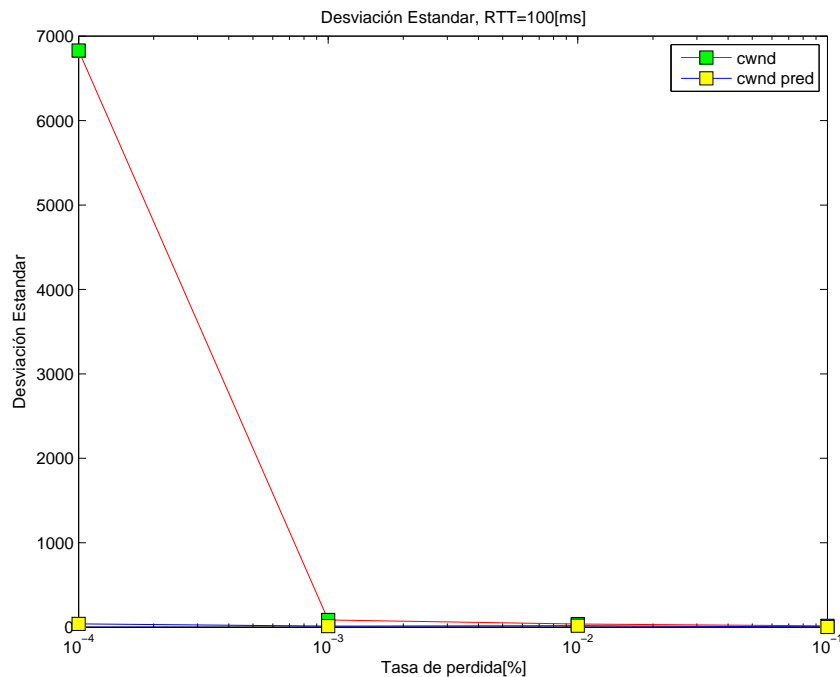


**Ilustración 75: Desviación estándar de la ventana de congestión para tcp\_reno para tasa de pérdida variable con RTT=10[ms]**

2)  $RTT=100[ms]$



**Ilustración 76: Media de la ventana de congestión para tcp\_reno para tasa de pérdida variable con RTT=100[ms]**



**Ilustración 77: Desviación estándar de la ventana de congestión para tcp\_reno para tasa de pérdida variable con RTT=100[ms]**

#### 4.3.3.3 Análisis de para módulo tcp\_reno con tasa de pérdida variable

Aun cuando los módulos implementados en el proyecto tienen una conducta oscilatoria en torno a la referencia queda en evidencia el comportamiento dispersivo de *tcp\_reno* presentando desviaciones estándar cuatro veces más elevadas. De esta forma se tiene alguna validez que insertando información al módulo, así como una referencia de control, es posible disminuir el comportamiento de ráfaga manteniendo una transferencia más estable.

En base a esto el comportamiento de los módulos del proyecto tienen validez y cumplen los objetivos planteados.

### 4.4 Resultados con valores del controlador PI variables

Esta prueba tiene como fin analizar cómo cambia el comportamiento del módulo *tcp\_conn\_pi* de acuerdo a los valores del controlador.

Se utilizan cuatro valores distintos para la constante proporcional del controlador PI y para cada una se tienen dos valores distintos de proporcionalidad  $R = \frac{K_P}{K_I}$ . Para  $K_P$  se utilizan valores de  $10^{-4}$ ,  $5 \cdot 10^{-4}$ ,  $10^{-5}$  y  $5 \cdot 10^{-5}$ . Dichos valores se eligieron porque fueron los que entregaron resultados más acordes a lo esperado en la simulación de OPNET. Para  $R$  se utilizan

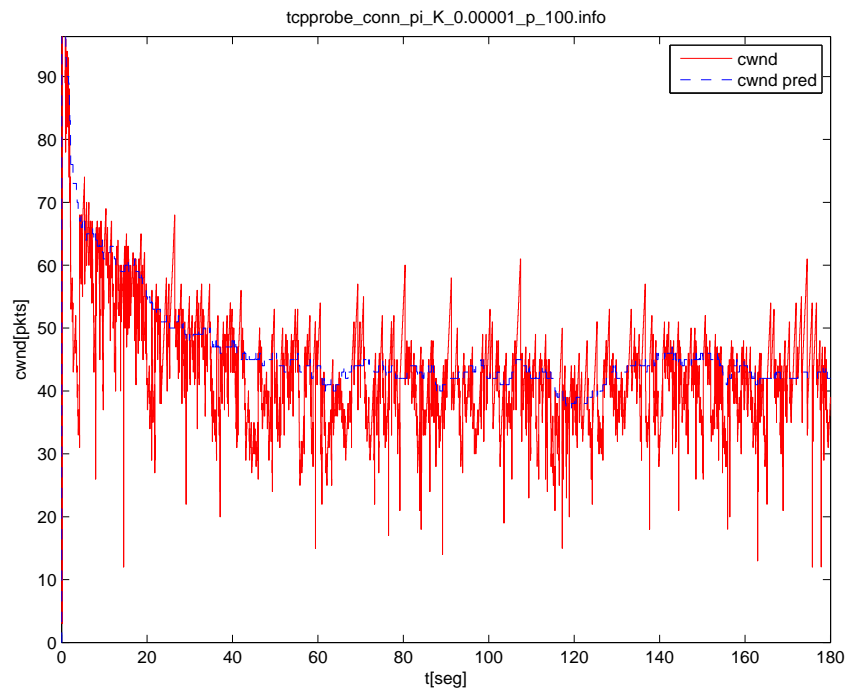
valores de 100 y 1000. Todas las siguientes pruebas tienen parámetros de  $RTT=10[ms]$ ,  $loss=0.1\%$  y un peso de la nueva estimación de  $\frac{1}{n}$ .

Esta prueba busca encontrar alguna relación entre el comportamiento del módulo respecto a cambios en el controlador. Por cambios se refiere principalmente a resultados estadísticos, es decir si tanto la media como la varianza se ven afectadas de acuerdo a los valores de controlador. Obviamente el mejor caso es donde la media de la ventana es más cercana a la referencia y en donde su varianza sea la menor posible.

#### 4.4.1.1 Gráficos de ventana de congestión vs tiempo

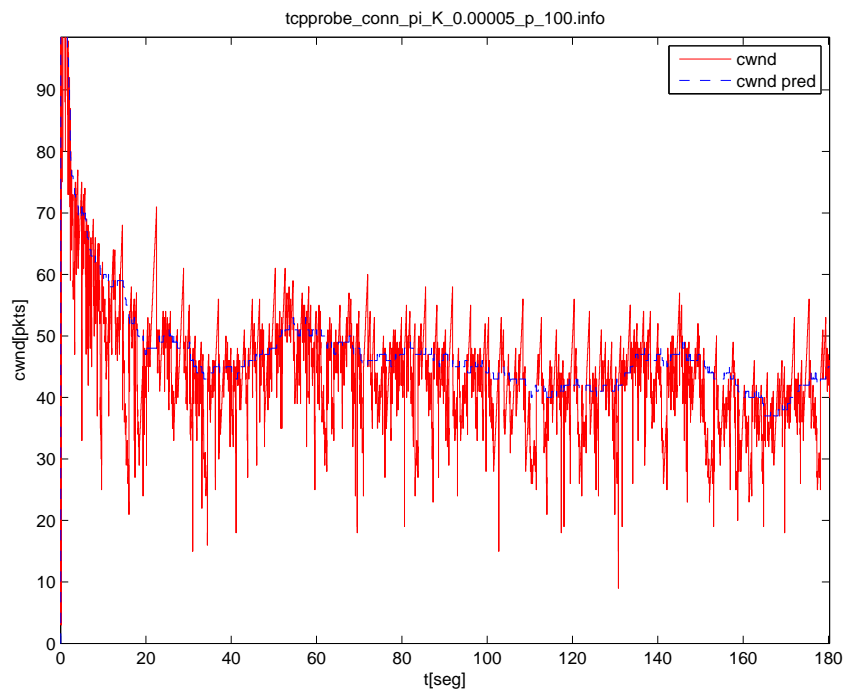
1)  $R=100$

Se utiliza primero un factor de  $R=100$ , es decir  $K_P$  es 100 veces más grande que  $K_I$ .

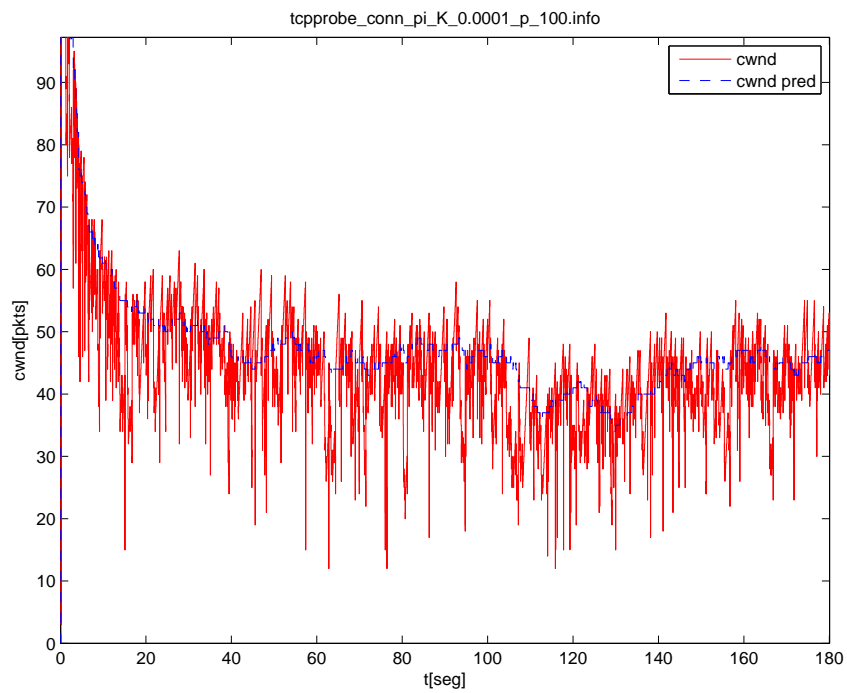


**Ilustración 78: Módulo tcp\_conn\_pi  $RTT=10[ms]$ ,  $Loss=0.1\%$ ,  $K_P=10^{-5}$ ,  $R=100$**

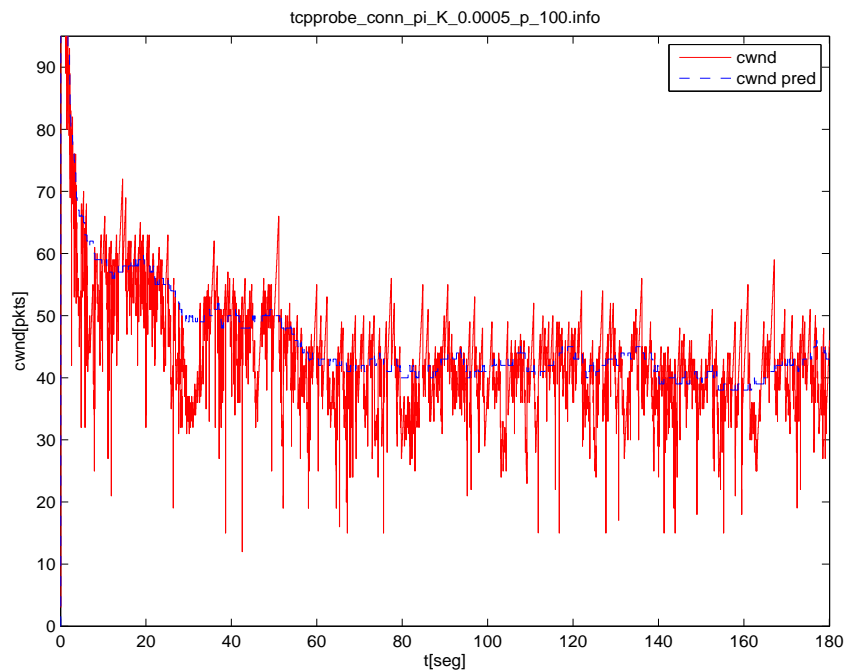




**Ilustración 79: Módulo tcp\_conn\_pi RTT=10[ms], Loss=0.1%, KP=5\*10<sup>-5</sup>, R=100**



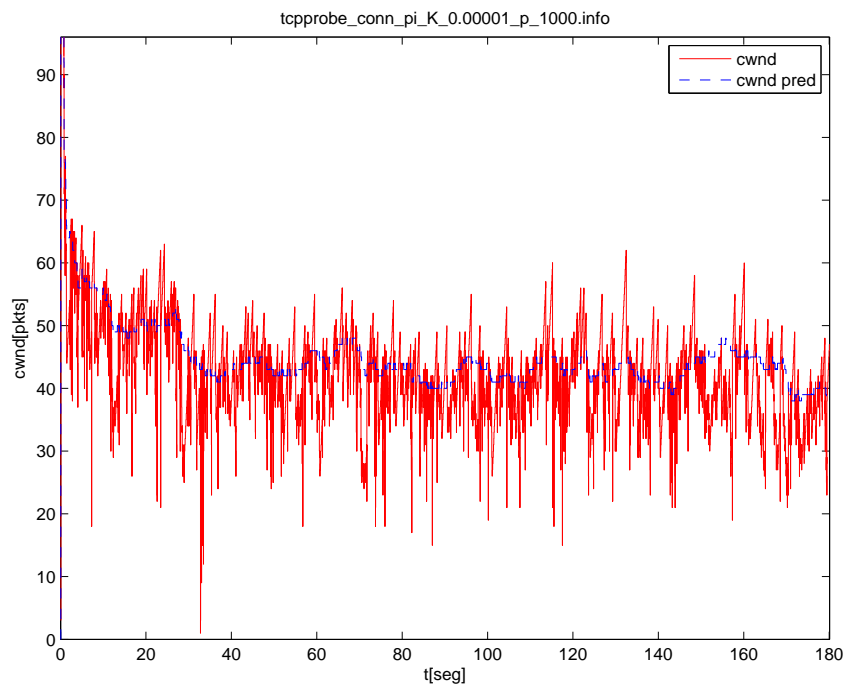
**Ilustración 80: Módulo tcp\_conn\_pi RTT=10[ms], Loss=0.1%, KP=10<sup>-4</sup>, R=100**



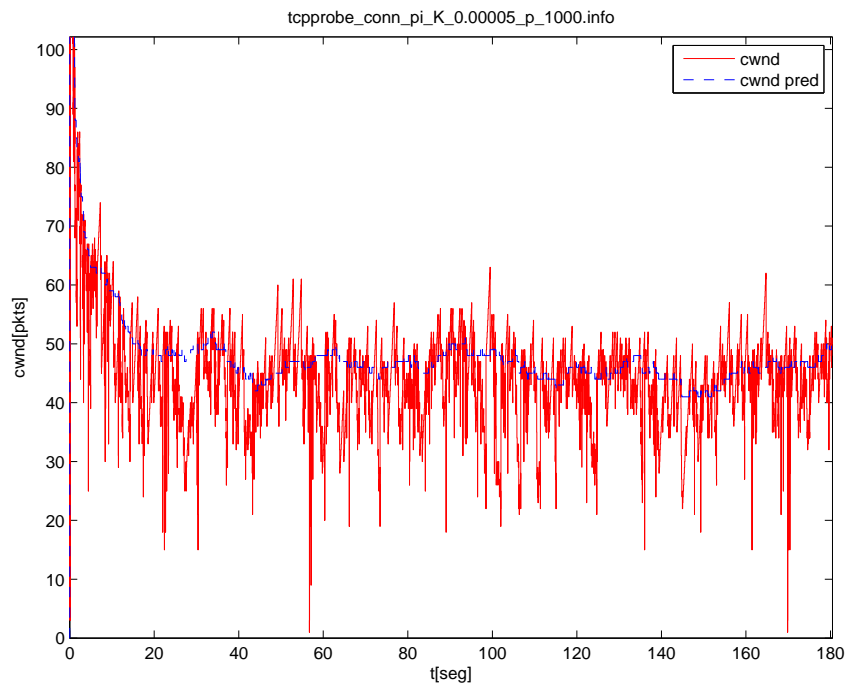
**Ilustración 81: Módulo tcp\_conn\_pi RTT=10[ms], Loss=0.1%,  $K_P=5 \cdot 10^{-4}$ , R=100**

2) R=1000

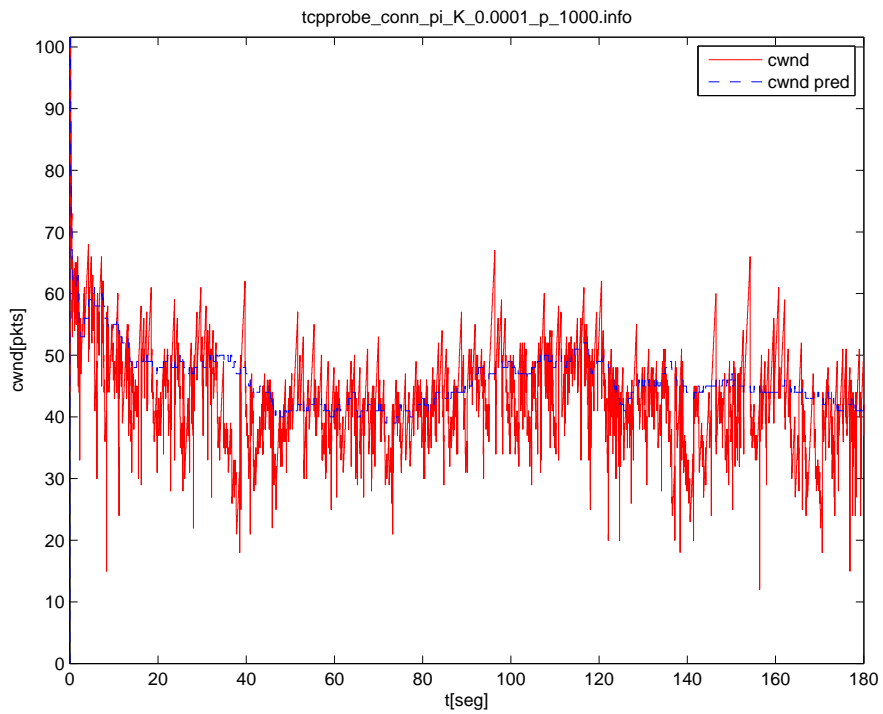
Se utiliza ahora un factor de R=1000, es decir  $K_P$  es 1000 veces más grande que  $K_I$  y se intentan obtener resultados diferentes.



**Ilustración 82: Módulo tcp\_conn\_pi RTT=10[ms], Loss=0.1%,  $K_P=10^{-5}$ , R=1000**



**Ilustración 83: Módulo tcp\_conn\_pi RTT=10[ms], Loss=0.1%, KP=5\*10<sup>-5</sup>, R=1000**



**Ilustración 84: Módulo tcp\_conn\_pi RTT=10[ms], Loss=0.1%, KP=10<sup>-4</sup>, R=1000**

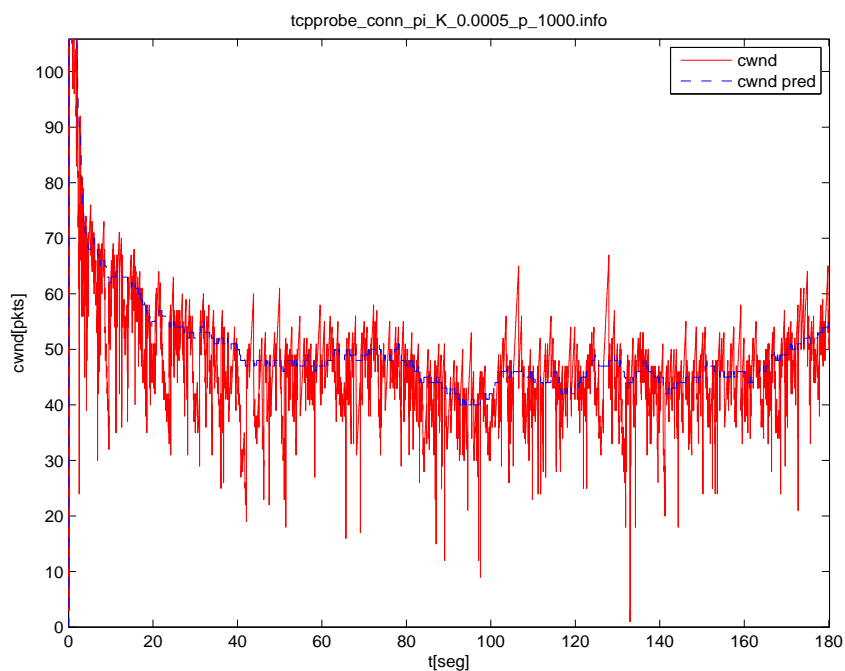


Ilustración 85: Módulo tcp\_conn\_pi RTT=10[ms], Loss=0.1%,  $K_P=5 \cdot 10^{-4}$ , R=1000

#### 4.4.1.2 Resultados estadísticos para el módulo tcp\_conn\_pi con $K_P$ variable

1) R=100

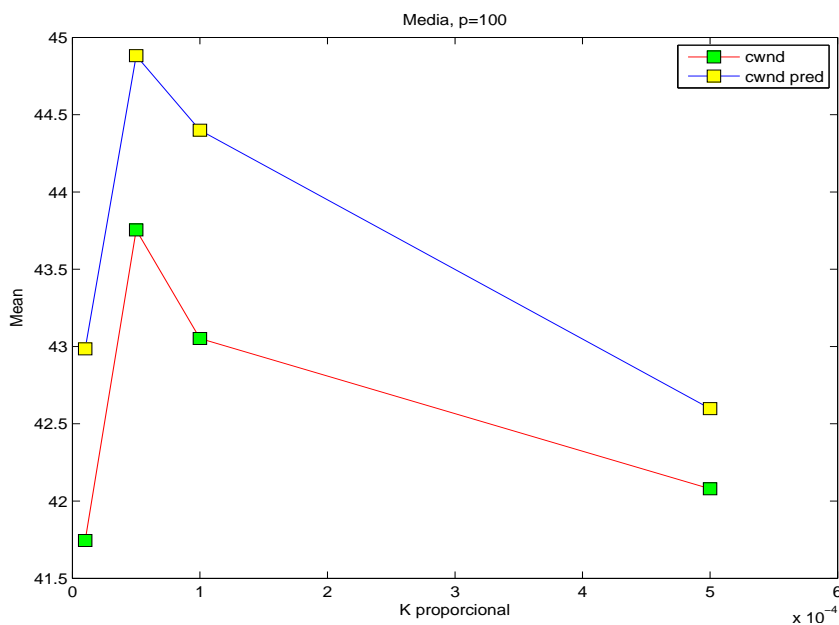
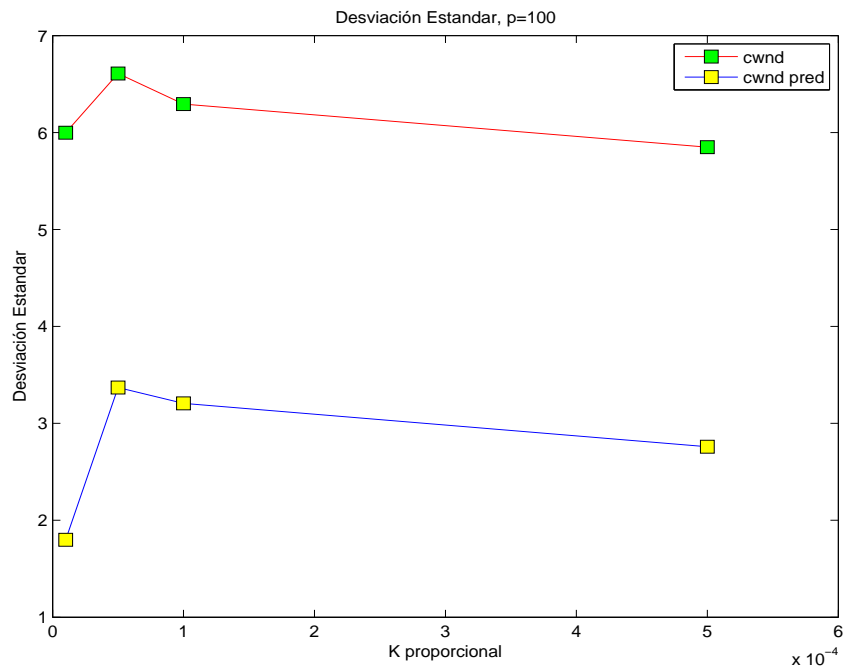
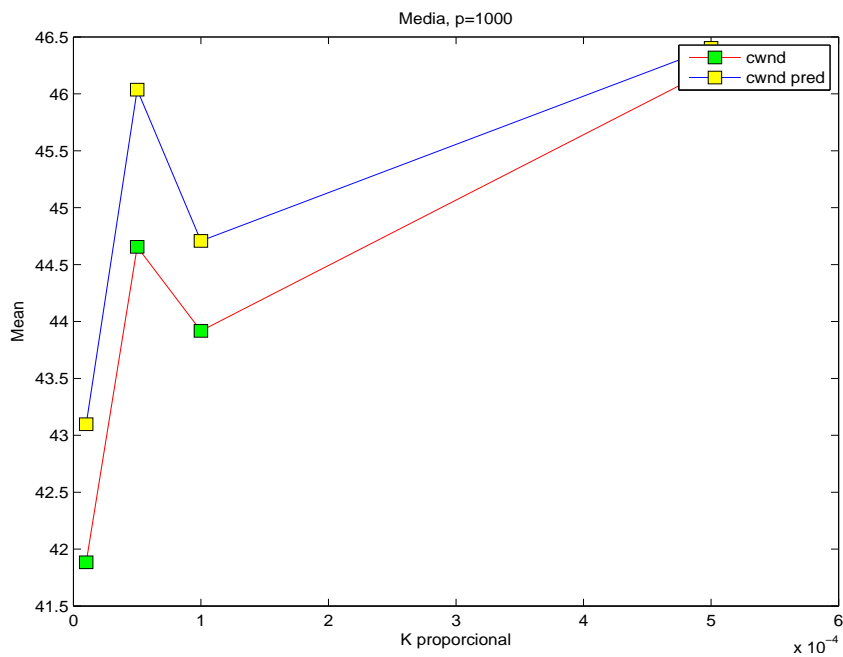


Ilustración 86: Media de la ventana de congestión para tcp\_conn\_pi para K variable con tasa de pérdida=0.1%, RTT=10[ms] y R=100

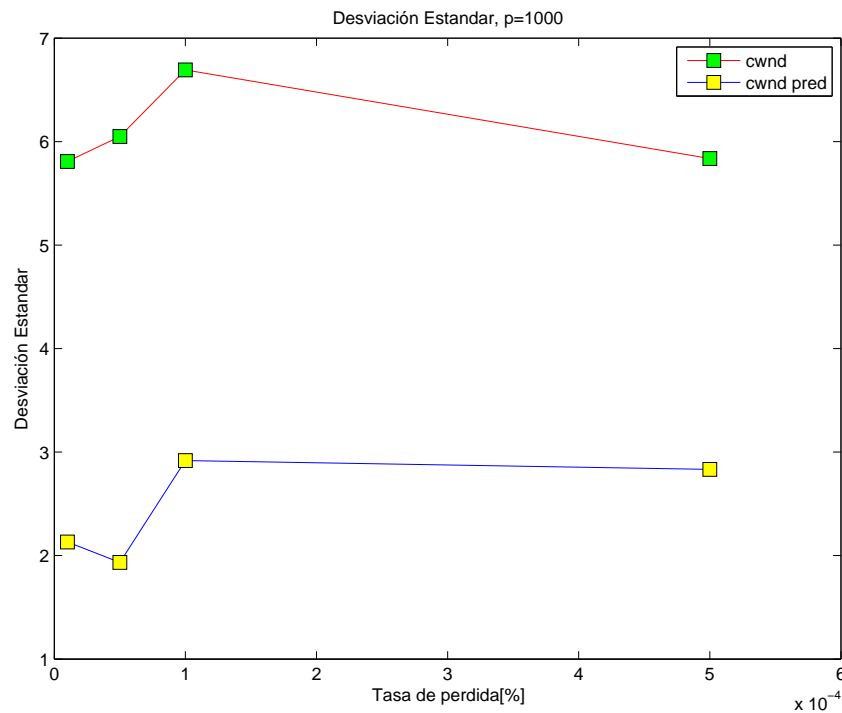


**Ilustración 87: Desviación estándar de la ventana de congestión para tcp\_conn\_pi para K variable con tasa de pérdida=0.1%, RTT=10[ms] y R=100**

2) R=1000



**Ilustración 88: Media de la ventana de congestión para tcp\_conn\_pi para K variable con tasa de pérdida=0.1%, RTT=10[ms] y R=1000**



**Ilustración 89: Desviación estándar de la ventana de congestión para tcp\_conn\_pi para  $K_P$  variable con tasa de pérdida=0.1%, RTT=10[ms] y  $R=1000$**

#### 4.4.1.3 Análisis de para módulo tcp\_conn\_pi con $K_P$ variable

Únicamente observando los gráficos de ventana de congestión vs tiempo no es posible encontrar alguna tendencia apreciable para los distintos valores del controlador por lo cual se utilizaran los resultados estadísticos para extraer alguna información relevante para esta sección.

A pesar de obtener diferentes resultados para cada  $K_P$  utilizado los valores difieren tan levemente que es mejor atribuirlos a variaciones sobre una media. Prácticamente las medias de la ventana de congestión oscilan entre 40[*pkts*] y 45[*pkts*]. Lo mismo ocurre para la desviación estándar en donde oscila en 1[*pkts*] y 7[*pkts*] en ambos casos independiente de la relación entre  $K_P$  y  $R$ . Por lo tanto son diferencias tan pequeñas para concluir sobre la inferencia de  $K_P$  y  $R$  en el comportamiento del módulo a menos con estos valores.

### 4.5 Resultados con valor de peso variable y tipo de peso

Para esta prueba, lo importante no es el valor que toma la ventana de congestión, sino la evolución de la ventana predicha y su utilidad para ser usada como referencia. Por lo mismo el algoritmo de evasión de congestión a utilizar no es importante y en este caso resulta indiferente al menos en el primer tipo de prueba cual utilizar (la tasa de pérdidas se fija por *NetEm*). Se presentan dos clasificaciones: una en donde el peso es inversamente proporcional a la cantidad

de pérdidas  $W \sim \frac{1}{n}$  con  $n$  como el contador, la otra clasificación se considera un peso fijo a las nuevas mediciones. Para el primer caso se utilizan pesos de  $\frac{1}{n}$ ,  $\frac{5}{n}$  y  $\frac{10}{n}$ . En el segundo caso se utilizan pesos de 1%, 5% y 10% respecto a la predicción precedente. Todas las pruebas se realizaron con parámetros de  $RTT$  igual a 10[ms] y una tasa de pérdida de 0.1%. Para este último caso se incluye en los gráficos la ventana de congestión puesto que se presenta un conflicto que será explicado más adelante.

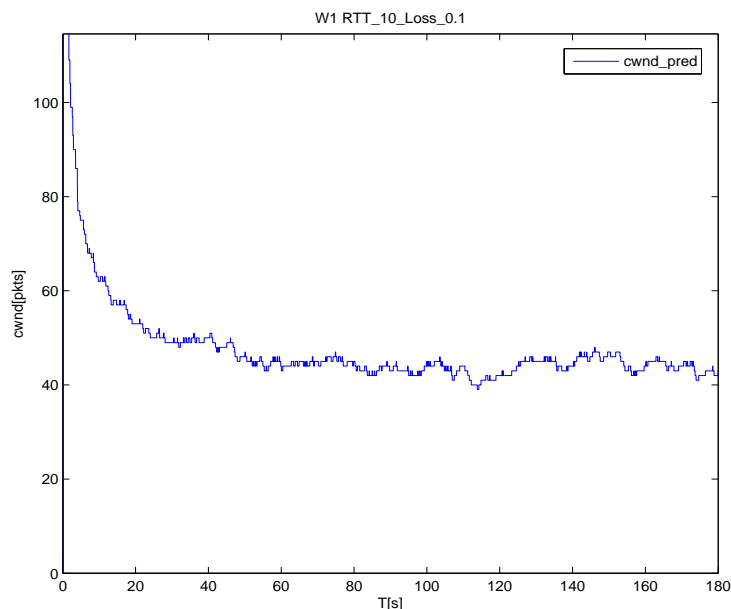
Estos resultados son importantes puesto que el tipo de peso puede influir de diversas maneras: valor de la capacidad del canal, velocidad a la que encuentra la capacidad del canal, desviación estándar de la capacidad canal encontrado y la más importante es si la referencia encontrada es válida.

Se espera que los pesos del tipo  $W \sim \frac{1}{n}$  tengan una varianza menor que los pesos porcentuales puesto que con el transcurso de la transmisión el valor de cada muestra para calcular la referencia disminuye.

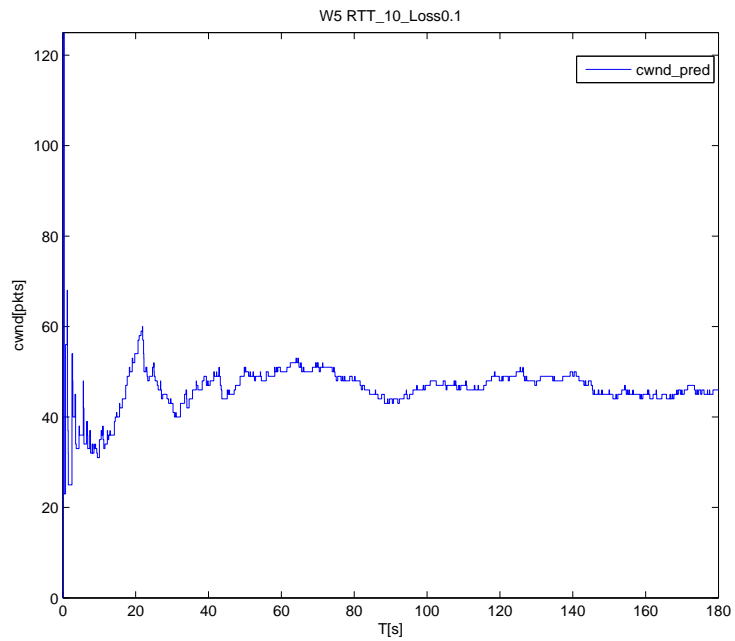
#### 4.5.1 Peso del tipo $W \sim \frac{1}{n}$

La ventana de congestión predicha corresponde a la capacidad del canal, por lo tanto no depende del módulo de evasión de congestión a utilizar. Utilizando este tipo de peso variable se busca encontrar como varía la referencia en cuando a velocidad y estadísticos.

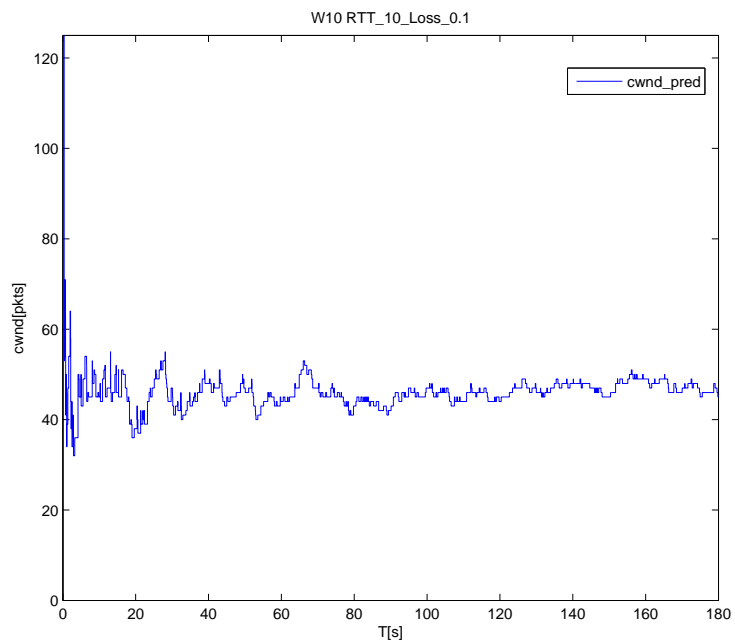
##### 4.5.1.1 Gráficos de ventana de congestión vs tiempo



**Ilustración 90:** `cwnd_pred` Módulo `tcp_conn_conn` con peso  $W=1$  y  $RTT=10$ [ms] y  $Loss=0.1\%$



**Ilustración 91: cwnd\_pred Módulo tcp\_conn\_conn con peso  $W=5$  y  $RTT=10$ [ms] y  $Loss=0.1\%$**

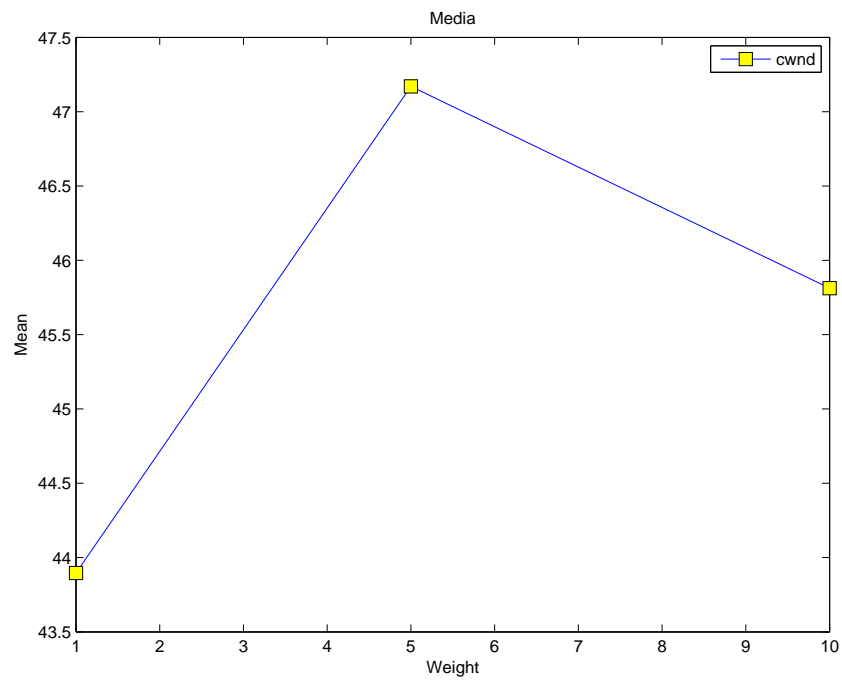


**Ilustración 92: cwnd\_pred Módulo tcp\_conn\_conn con peso  $W=10$  y  $RTT=10$ [ms] y  $Loss=0.1\%$**

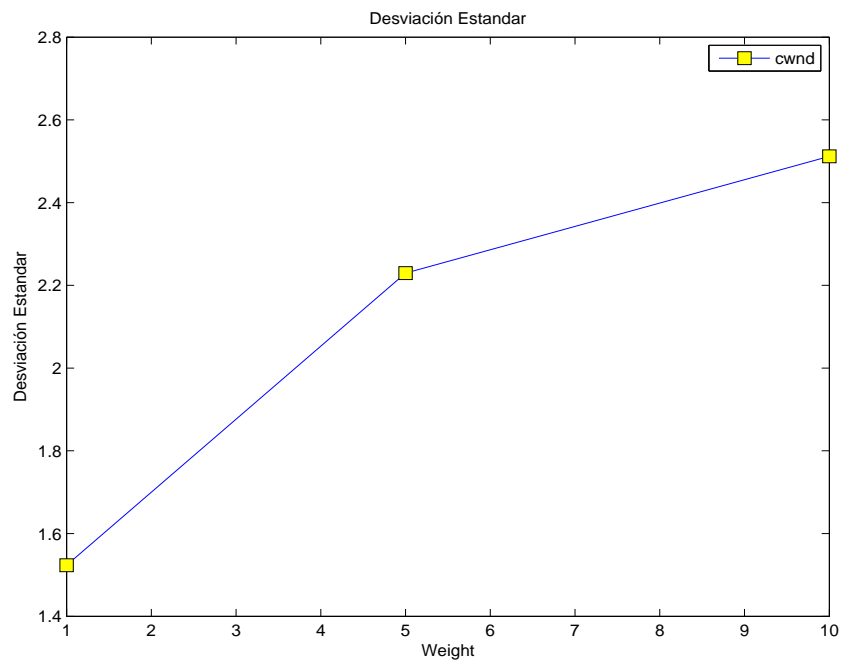
Se observa que a mayor peso la ventana predicha se encuentra más rápido.



### 4.5.1.2 Resultados estadísticos para peso variable del tipo $W \sim \frac{1}{n}$



**Ilustración 93: Media de la ventana de congestión predicha para tcp\_conn\_alt para peso W variable con tasa de pérdida=0.1% y RTT=10[ms]**



**Ilustración 94: Desviación estándar de la ventana de congestión predicha para tcp\_conn\_alt para peso W variable con tasa de pérdida=0.1% y RTT=10[ms]**

### 4.5.1.3 Análisis de convergencia de la ventana predicha con peso variable del tipo $W \sim \frac{1}{n}$

Para los tres valores de peso utilizados se obtuvieron medias muy parecidas lo cual indica que el valor del peso no afecta en gran medida el resultado de la predicción de la ventana de congestión obtenida. Sin embargo la desviación estándar aumenta a medida que el peso también sube. Esto indica que la última muestra de la ventana para estimar  $\Delta$  al tener un peso mayor produce oscilaciones y menos estabilidad respecto a la media. La razón por la cual utilizar un peso más alto a pesar de sacrificar una disminución en la desviación estándar quedara expresada en la siguiente sección donde la capacidad del canal se ve alterada.

## 4.5.2 Peso constante

Es este caso utilizando un peso porcentual fijo para cada nueva muestra, dado el tipo de resultado encontrado también se grafica la ventana de congestión además de la predicción.

### 4.5.2.1 Gráficos de ventana de congestión vs tiempo

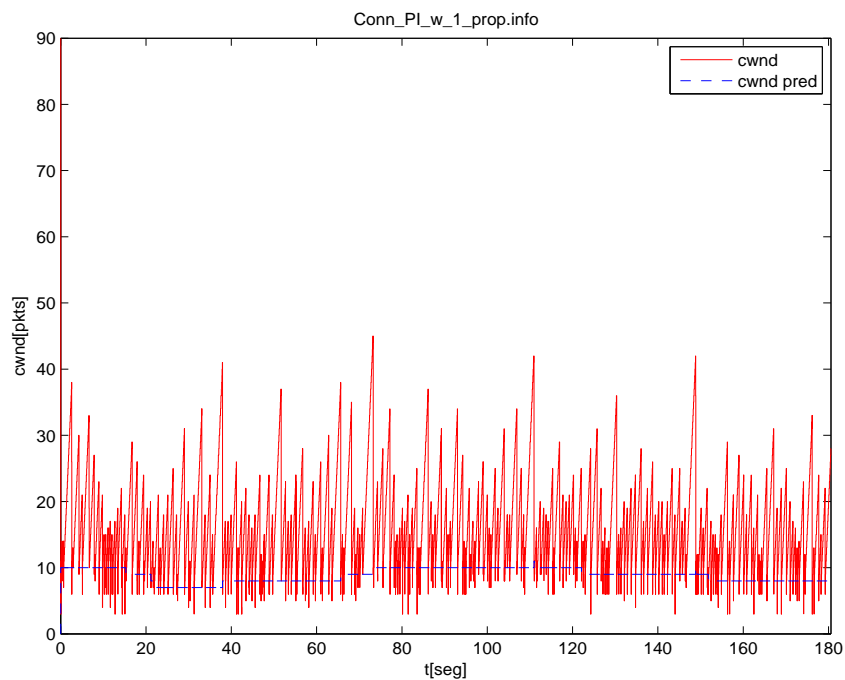
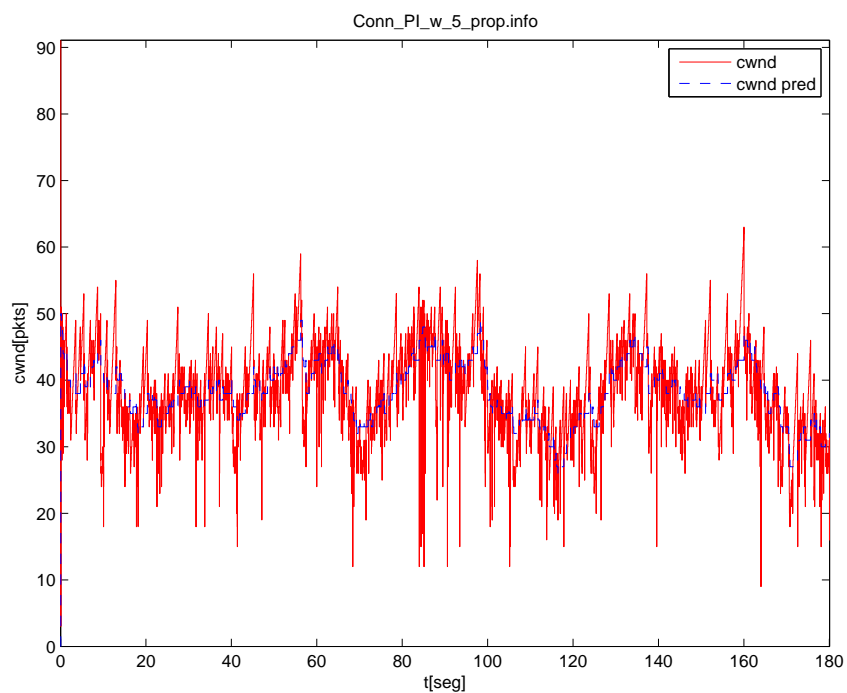
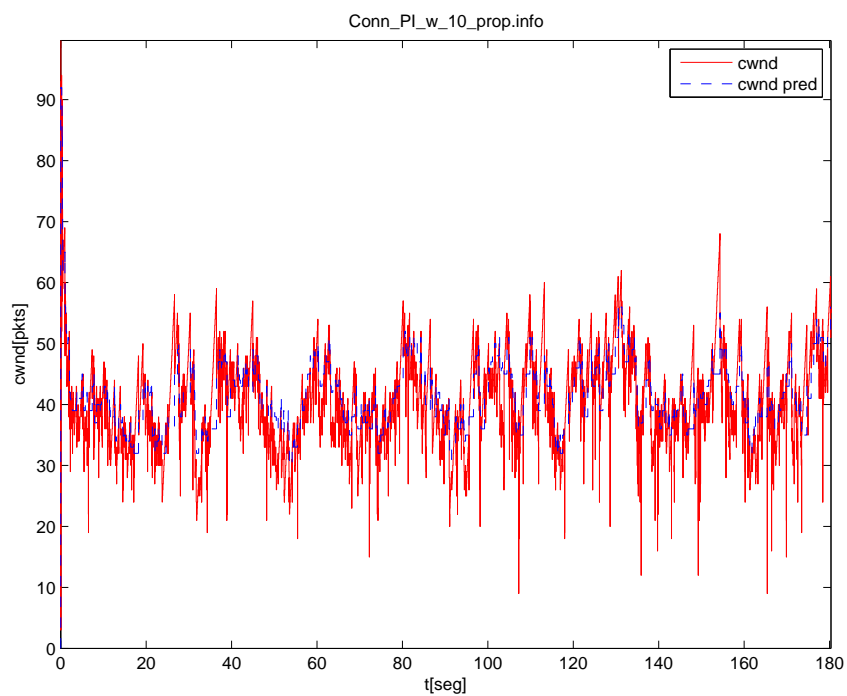


Ilustración 95: Módulo tcp\_conn\_pi RTT=10[ms], Loss=0.1% y peso proporcional W=1%



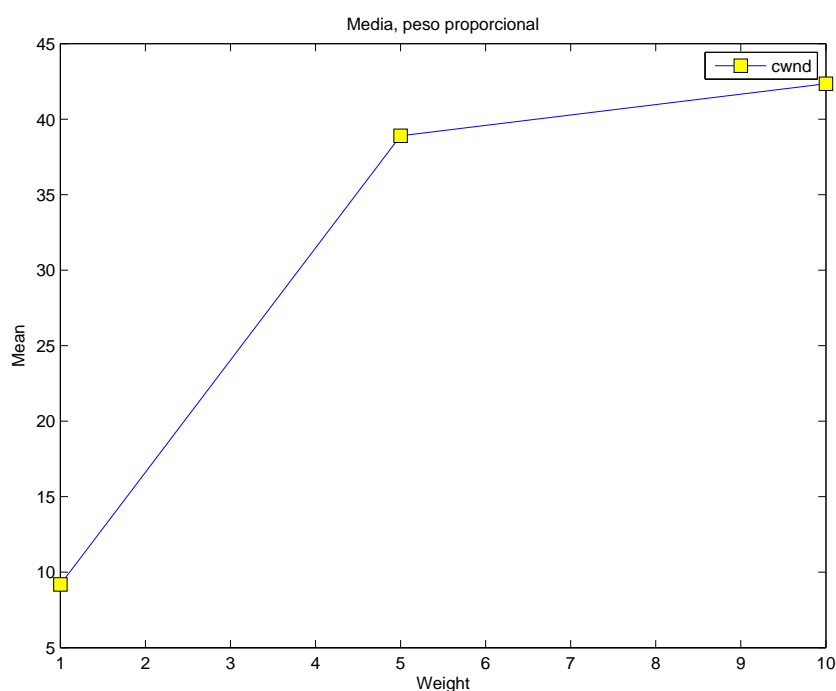
**Ilustración 96: Módulo tcp\_conn\_pi RTT=10[ms], Loss=0.1% y peso proporcional W=5%**



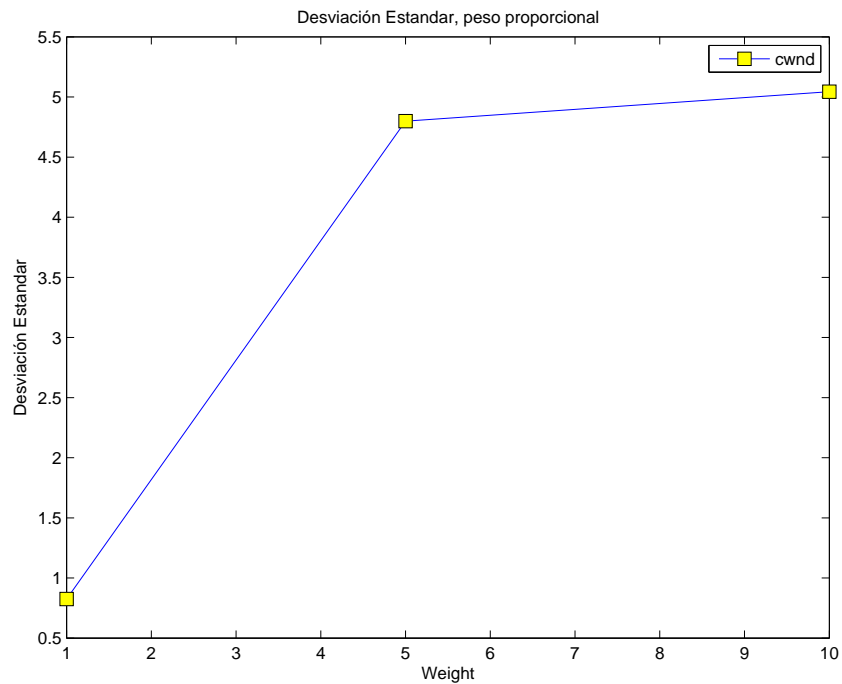
**Ilustración 97: Módulo tcp\_conn\_pi RTT=10[ms], Loss=0.1% y peso proporcional W=10%**

Utilizando un peso fijo para las nuevas muestras ingresantes para el cálculo de la ventana predicha ocurre un fenómeno en donde la predicción intenta seguir la ventana de congestión, razón por la cual también se incluye esta en los gráficos. Cuando el peso corresponde al 1% prácticamente la predicción se estanca y nunca llega a la capacidad del canal. Con pesos de 5% y 10% la predicción sigue a la ventana de congestión en el valor donde hubo pérdida dado el  $\Delta$  obtenido, a pesar de que la desviación estándar es baja, esta no es un indicador de convergencia y la explicación para esto es simplemente que la ventana de congestión predicha oscila sobre la verdadera capacidad del canal. Este comportamiento es en principio contraproducente: la idea prima del proyecto es que la ventana de congestión tienda a la predicción y no al revés. Otra forma de explicarlo es que la predicción no dependa del algoritmo de evasión de congestión a utilizar.

#### 4.5.2.2 Resultados estadísticos para peso variable del tipo proporcional



**Ilustración 98: Media de la ventana de congestión predicha para tcp\_conn\_pi para peso W proporcional variable con tasa de pérdida=0.1% y RTT=10[ms]**



**Ilustración 99: Desviación estándar de la ventana de congestión predicha para tcp\_conn\_pi para peso W proporcional variable con tasa de pérdida=0.1% y RTT=10[ms]**

### 4.5.2.3 Análisis de convergencia de la ventana predicha con peso proporcional

Como se explicó previamente utilizando un peso de 1% no se encuentra la capacidad del canal. Utilizando pesos de 5% y 10% se encuentra una referencia con medias muy parecidas indicando que cualquiera de los dos valores de pesos utilizados funcionan bien bajo estos parámetros.

En cuanto a la desviación estándar se repite el fenómeno observado en el peso  $W \sim \frac{1}{n}$ , en donde ésta aumenta a medida que el peso aumenta.

En base a los resultados obtenidos de acuerdo a valores de media, desviación estándar y convergencia el mejor peso resulta ser  $\frac{1}{n}$ .

## 4.6 Resultados con cambios en la capacidad del canal durante la transmisión

En esta prueba solo se utilizara el módulo *tcp\_conn\_pi* para evaluar resultados y el modulo reno para el análisis comparativo, al igual que en la prueba anterior se varia el valor y tipo de peso a utilizar para la estimación de la ventana de congestión, sin embargo este análisis tiene un objetivo distinto y corresponde a evaluar la reacción del módulo de acuerdo al peso cuando la capacidad del canal aumenta abruptamente, es decir, la tasa de pérdida se reduce. Se dividirá en dos partes, la primera usando el peso  $W \sim \frac{1}{n}$  para valores de 1 y 10 y la segunda un peso porcentual de  $W = 5\%$ . La capacidad del canal cambia drásticamente a los 100[s] de transmisión, el *RTT* en ambos casos es de 10[ms] y la primera capacidad del canal tiene una tasa de pérdida de paquetes de 0.1% y aumenta hasta 0.001%.

Esta prueba tiene como objetivo estudiar la reacción del cálculo de la predicción de la capacidad del canal cuando ésta cambia. Esto es importante pues se busca encontrar el mejor peso que resuelva este fenómeno.

Se espera a priori que el mejor resultado sea para el peso de 5% pues este tipo de peso no depende del instante en donde la capacidad del canal varía.

### 4.6.1.1 Gráficos de ventana de congestión vs tiempo

1) Peso del tipo  $W \sim \frac{1}{n}$

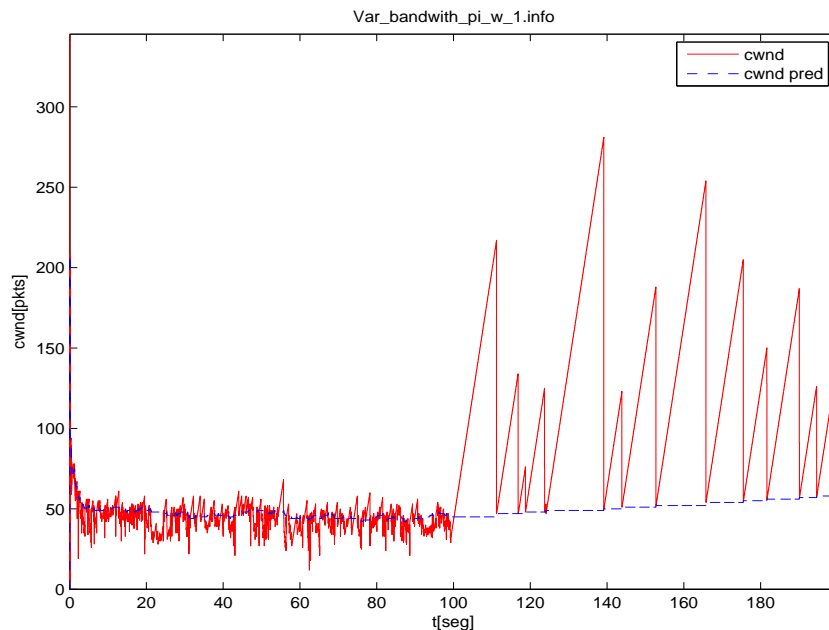
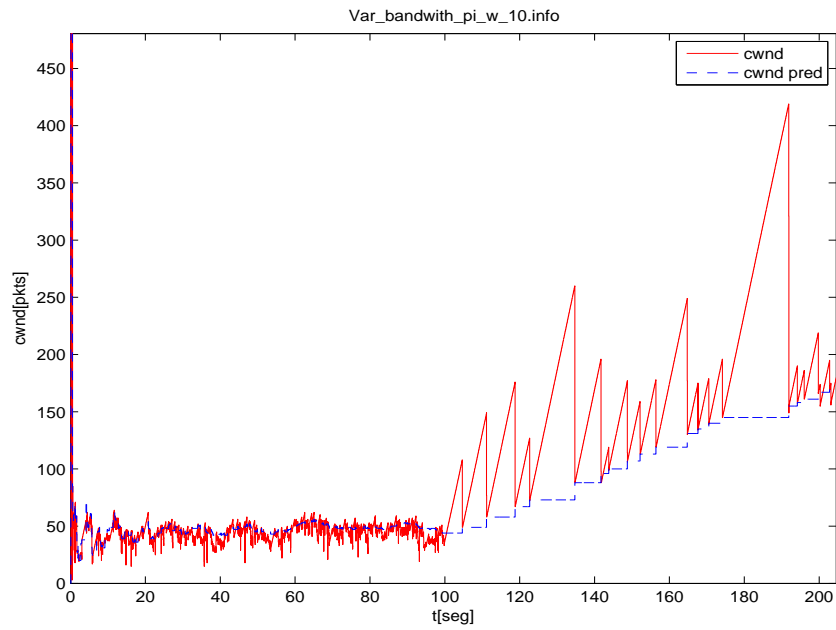
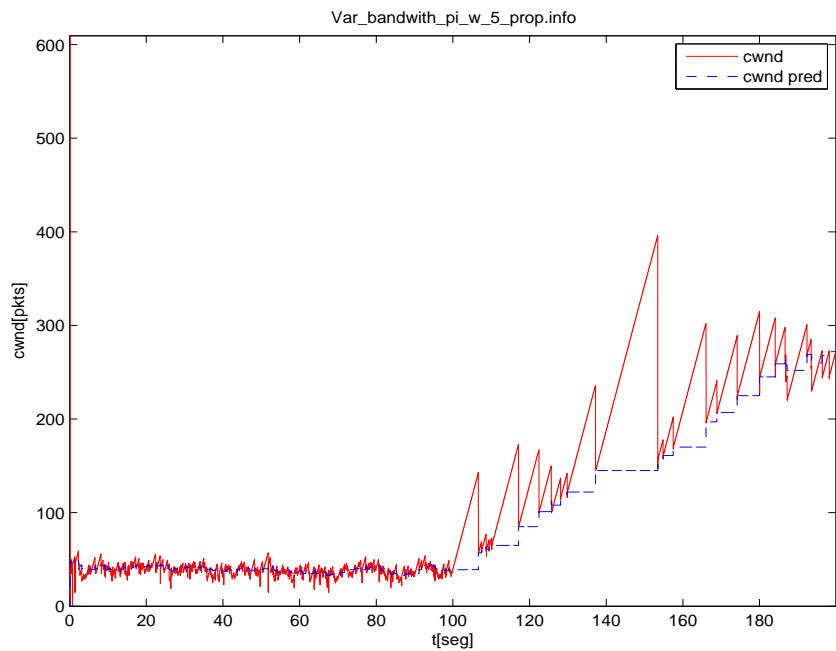


Ilustración 100: Módulo *tcp\_conn\_pi* RTT=10[ms] con capacidad del canal cambiante de Loss=0.1% a 0.001% y peso  $W=1$



**Ilustración 101: Módulo tcp\_conn\_pi RTT=10[ms] con capacidad del canal cambiante de Loss=0.1% a 0.001% y peso W=10**

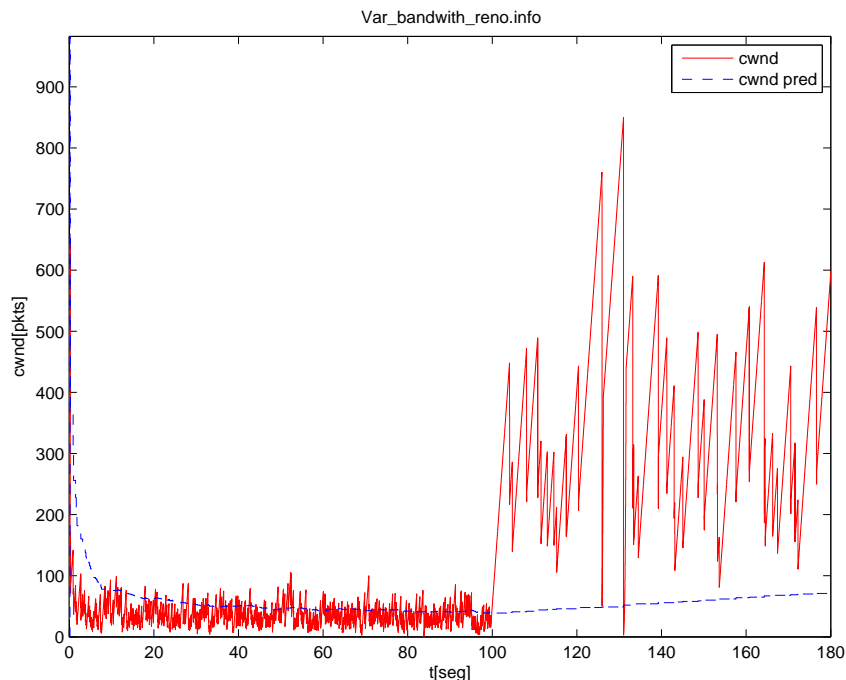
2) Peso constante de 5%



**Ilustración 102: Módulo tcp\_conn\_pi RTT=10[ms] con capacidad del canal cambiante de Loss=0.1% a 0.001% y peso proporcional W=5%**

En esta prueba se aprecia que la mejor predicción obtenida con el peso de la última muestra igual a  $\frac{1}{n}$  tiene una muy mala reacción ante un cambio en la capacidad del canal debido que una vez converge a un valor se estabiliza allí porque las muestras subsiguientes cada vez pierden valor. Aumentando dicho peso a  $\frac{10}{n}$  se presenta una mejor reacción sin embargo el problema radica en que el peso del tipo  $W \sim \frac{1}{n}$  tiende eventualmente a perder el valor de las muestras acorde la transmisión avanza, si la conexión es lo suficientemente larga y eventualmente la capacidad del canal se ve alterada es posible que dado el caso un peso de incluso  $\frac{100}{n}$  sea insuficiente. Por el contrario, el peso proporcional a la cantidad de muestras de 5% utilizado en esta prueba entrega resultados favorables dado que presenta una rápida reacción al cambio de capacidad del canal.

### 3) Modulo Reno



**Ilustración 103: Módulo tcp\_reno RTT=10[ms] con capacidad del canal cambiante de Loss=0.1% a 0.001% y peso W=1**

Utilizando Reno, que es independiente de la capacidad del canal, su *throughput* aumenta inmediatamente ocurre la reducción de la tasa de pérdida.

#### 4.6.1.2 Análisis de reacción de la predicción para distintos valores y tipos de peso

Estos resultados producen un dilema entre qué tipo de peso a utilizar: el peso  $\frac{1}{n}$  que logra que la predicción converja a la capacidad del canal sin producción oscilaciones e



independiente del algoritmo de evasión a utilizar pero con una reacción lenta o nula para casos en donde la capacidad del canal varia, o el peso fijo de 5% en donde la predicción no converge en el tiempo sino más bien oscila en la capacidad del canal y se ve alterada en cierta medida por el comportamiento del módulo pero que tiene una buena reacción cuando la capacidad del canal cambia. La respuesta depende del tipo de canal a utilizar, es muy raro que la capacidad cambie por lo cual en la mayoría de los casos bastaría con el peso  $\frac{1}{n}$  para una predicción más precisa.

## 4.7 Tabla resumen de estadísticas

### 4.7.1 RTT variable

**Tabla 2: Media de la ventana de congestión para tasa de pérdida de 0.1%**

RTT[ms]		10	20	30	40	50	60	70	80	90	100
Tcp_Reno	44,845	48,899	45,519	43,523	44,034	46,979	44,331	43,901	42,107	35,555	45,371
Predicción Reno	44,297	45,306	43,157	54,283	49,282	56,460	58,954	56,335	52,633	50,042	64,331
Tcp_conn_alt	38,301	37,959	38,68	38,86	41,19	41,12	42,973	39,268	43,934	40,952	38,182
Predicción Tcp_conn_alt	44,899	43,294	45,519	49,224	49,499	55,087	64,644	53,449	56,517	59,349	52,828
Tcp_conn_pi	46,187	42,426	46,639	48,193	52,414	52,507	52,595	55,399	64,212	54,437	59,076
Predicción Tcp_conn_pi	44,468	43,693	46,485	46,719	50,710	50,784	51,800	54,585	63,570	53,830	58,851

**Tabla 3: Desviación estándar de la ventana de congestión para tasa de pérdida de 0.1%**

RTT[ms]	1	10	20	30	40	50	60	70	80	90	100
Tcp_Reno	18,523	20,243	18,574	17,580	20,836	21,139	18,507	19,982	17,542	13,669	19,203
Predicción Reno	2,681	1,694	2,841	4,912	2,873	6,200	6,468	4,134	2,298	4,717	4,882
Tcp_conn_alt	6,526	6,591	7,577	5,788	7,005	7,098	8,426	8,652	7,821	8,013	8,048
Predicción Tcp_conn_alt	2,859	1,737	4,984	1,959	2,830	2,243	5,080	5,906	2,718	6,540	5,396
Tcp_conn_pi	5,177	5,729	4,638	5,664	5,659	5,159	5,207	4,464	9,398	5,916	7,483
Predicción Tcp_conn_pi	2,544	2,588	2,278	4,182	2,421	3,454	3,759	2,877	8,364	2,975	6,077

**Tabla 4: Media de la ventana de congestión para tasa de pérdida de 0.001%**

RTT[ms]	1	10	20	30	40	50	60	70	80	90	100
Tcp_Reno	44,230	389,207	470,317	431,914	562,213	409,536	728,673	516,022	1124,044	464,505	373,033
Predicción Reno	44,587	424,698	506,069	544,213	513,091	441,677	574,159	544,857	641,052	520,594	404,588
Tcp_conn_alt	430,094	380,451	376,532	378,368	391,058	468,454	493,121	474,732	394,118	309,519	554,517
Predicción Tcp_conn_alt	467,413	413,073	437,400	479,191	466,418	469,696	436,020	458,317	394,426	305,312	475,560
Tcp_conn_pi	493,371	483,318	370,388	539,809	468,346	598,013	470,463	524,151	450,505	392,122	519,326
Predicción Tcp_conn_pi	467,404	453,241	389,973	506,033	431,614	552,261	444,721	508,337	421,324	321,307	501,861

**Tabla 5: Desviación estándar de la ventana de congestión para tasa de pérdida de 0.001%**

RTT[ms]	1	10	20	30	40	50	60	70	80	90	100
Tcp_Reno	17,989	137,015	162,424	147,582	184,244	149,175	224,665	149,537	809,006	213,315	112,682
Predicción Reno	2,952	6,031	17,178	35,376	17,408	24,862	8,013	15,332	41,362	43,083	12,229
Tcp_conn_alt	54,570	71,835	42,545	42,540	55,370	39,130	45,266	92,593	40,375	31,084	167,494
Predicción Tcp_conn_alt	6,842	10,941	12,031	19,737	26,010	11,166	12,396	22,258	8,287	7,310	73,167
Tcp_conn_pi	32,265	28,551	54,635	42,222	35,771	51,577	36,407	50,224	28,633	56,241	67,885
Predicción Tcp_conn_pi	16,113	7,694	13,051	18,590	32,014	21,017	25,180	47,037	18,432	7,117	62,090

## 4.7.2 Tasa de pérdida variable

**Tabla 6: Media de la ventana de congestión para RTT=10[ms]**

Tasa de pérdida	0.1	0.01	0.001	0.0001
Tcp_Reno	1003,493	458,794	150,512	44,922
Predicción Reno	630,853	486,366	149,404	43,984
Tcp_conn_alt	973,369	372,511	125,249	38,185
Predicción Tcp_conn_alt	674,173	451,280	144,143	44,573
Tcp_conn_pi	735,447	473,155	135,562	43,779
Predicción Tcp_conn_pi	604,364	447,069	141,294	44,333

**Tabla 7: Desviación estándar de la ventana de congestión para RTT=10[ms]**

Tasa de pérdida [%]	0.1	0.01	0.001	0.0001
Tcp_Reno	273,431	168,971	60,651	17,895
Predicción Reno	17,365	11,496	5,624	2,546
Tcp_conn_alt	42,314	43,873	18,712	6,130
Predicción Tcp_conn_alt	54,290	15,685	3,584	2,180
Tcp_conn_pi	79,873	28,184	16,385	6,349
Predicción Tcp_conn_pi	64,343	13,462	2,022	2,568

**Tabla 8: Media de la ventana de congestión para RTT=100[ms]**

Tasa de pérdida [%]	0.1	0.01	0.001	0.0001
Tcp_Reno	5863,382	332,682	130,047	43,355
Predicción Reno	610,400	380,302	206,056	60,010
Tcp_conn_alt	933,506	388,616	161,591	41,290
Predicción Tcp_conn_alt	61,806	414,238	174,910	67,284
Tcp_conn_pi	914,468	535,398	192,363	58,959
Predicción Tcp_conn_pi	891,764	514,680	186,557	58,313

**Tabla 9: Desviación estándar de la ventana de congestión para RTT=100[ms]**

Tasa de pérdida [%]	0.1	0.01	0.001	0.0001
Tcp_Reno	6830,407	85,428	37,845	16,439
Predicción Reno	38,204	11,941	18,072	3,238
Tcp_conn_alt	44,713	25,413	24,499	9,004
Predicción Tcp_conn_alt	8,726	3,937	3,111	6,176
Tcp_conn_pi	38,690	35,915	10,418	6,193
Predicción Tcp_conn_pi	61,285	28,912	5,661	3,799

**Tabla 10: Media y Desviación estándar de la ventana de congestión para KP variable**

K <sub>p</sub>	0.0001	0.0005	0.00001	0.00005
Media R=100	41,745	43,755	43,052	42,079
Media predicción R=100	42,985	44,882	44,400	42,598
Desviación estándar R=100	5,998	6,609	6,293	5,849
Desviación estándar predicción R=100	1,799	3,370	3,207	2,759
Media R=1000	41,884	44,656	43,917	46,216
Media predicción R=1000	43,097	46,037	44,707	46,404
Desviación estándar R=1000	5,808	6,049	6,692	5,836
Desviación estándar predicción R=1000	2,131	1,934	2,917	2,833

**Tabla 11: Media y Desviación estándar de la ventana predicha para pesos variables**

Peso	1/n	5/n	10/n	1%	5%	10%
Media predicción	43,897	47,170	45,812	9,191	38,895	42,342
Desviación estándar predicción	1,524	2,229	2,512	0,825	4,799	5,044

# CAPÍTULO 5: CONCLUSIONES Y TRABAJO FUTURO

## 5.1 Introducción

En este último capítulo se presentan las conclusiones obtenidas de las pruebas del Capítulo 4 de Análisis de Resultados y también se plantean nuevas propuestas que son posibles realizar en base a los resultados encontrados manteniendo el contexto de la misma línea de investigación en capa de transporte.

## 5.2 Conclusiones

El objetivo del proyecto se cumplió en lo referente a encontrar la capacidad del canal mediante el uso de la distancia entre paquetes perdidos consecutivos denominada  $\Delta$  para realizar la estimación de la tasa de pérdida presente en la transmisión. En lo referente a los módulos implementados, *tcp\_conn\_alt* y *tcp\_conn\_pi*, que utilizan el valor de la capacidad del canal para su funcionamiento tienen un rendimiento bajo lo esperado considerando las limitaciones adherentes al sistema operativo.

La restricción impuesta en Linux en donde la ventana de congestión no puede ser invariante en el tiempo dentro del módulo indica la no convergencia de ésta a la capacidad del canal resulta en un problema que en principio puede suponer el incumplimiento de los objetivos, sin embargo, cambiando el propósito de converger a la capacidad del canal a uno más bien de mantenerse lo más cercano posible a ésta con la menor cantidad y amplitud de oscilaciones posibles de manera estable soluciona esta problemática.

El primer módulo implementado, *tcp\_conn\_alt*, presenta un comportamiento esperado dado que se utiliza un controlador proporcional con algunas características particulares tales como que  $\alpha_{\min}$  es de 0.16 y no 0, el controlador solo actúa cuando la variable controlada está bajo la ventana predicha y en la fase de incremento de la ventana, sobre la referencia se utiliza forzosamente  $\alpha_{\min}$ , para instantes de pérdida bajo la capacidad del canal se utiliza la relación de *fairness* entre  $\alpha$  y  $\beta$  y sobre la predicción se fuerza a ser ésta misma. Todos estos factores producen un comportamiento oscilante alrededor del valor de la capacidad encontrada del canal, sin embargo en términos generales la media se encuentra bajo la predicción por tratarse de un controlador proporcional.

El segundo módulo, *tcp\_conn\_pi*, se desprende del primero y corresponde a una actualización en donde se agrega un componente de integración por lo tanto el controlador utilizado en este caso corresponde a un PI con las mismas características particulares de *tcp\_conn\_alt*. El comportamiento oscilatorio es esperado dado que aunque se tratase de un controlador PI, el sistema operativo fuerza a mantener la ventana de congestión invariante en el

tiempo, sin embargo en términos de media existe una clara convergencia: la media de la ventana de congestión para este módulo resulta en valores bastante cercanos a predicción obtenida.

El comportamiento de ambos módulos a pesar de presentar oscilaciones, éstas son incomparables a las del módulo *tcp\_reno* y es posible decir en base a los resultados estadísticos obtenidos que se cumple el propósito de mantenerse establemente en la ventana de congestión predicha por lo tanto el objetivo está cumplido.

Sobre los resultados obtenidos para pruebas de *RTT* resulta que el retardo no afecta la desviación estándar de la ventana de congestión. Los módulos implementados se basan en los instantes de pérdida de paquetes como estimador de congestión y por lo tanto el *RTT* resulta ser una variable indiferente para los resultados. Por muy diferentes que pareciesen los resultados de pruebas en un caso utilizando un *RTT* bajo y en otro un *RTT* elevado, en realidad el primero corresponde a una versión comprimida en el tiempo del segundo, es decir, el retardo no altera el comportamiento del módulo sino que ensancha el temporalmente el comportamiento de la ventana de congestión.

Por otra parte, el valor de la tasa de pérdida de paquetes, a diferencia del valor del retardo, tiene una implicancia directa en los comportamientos de los módulos y además de la predicción misma de la capacidad del canal. Lo cual se demuestra en como la ventana de congestión se ve alterada en cuanto a las amplitudes de esta. En cuanto a la predicción misma, dado que ésta requiere de muestras de instantes de pérdida de paquetes para obtener  $\Delta$  y así obtener una muestra de la capacidad, la frecuencia de las pérdidas afecta directamente la velocidad de convergencia, lo cual no sería problema si la transmisión es lo suficientemente larga.

No se obtuvo información suficiente para analizar cuanto afectan las constantes del controlador PI a la conducta del módulo respectivo. Esto se debe en parte a la dificultad para elegir un valor inicial (que en el presente caso se obtuvo luego de realizar diversas simulaciones en OPNET). Por lo cual se sugiere como trabajo futuro un análisis de dichos parámetros dado que la referencia del sistema ya existe y se asume por los resultados obtenidos que es correcta.

Un resultado interesante es la reacción obtenida para la ventana predicha de acuerdo al valor y tipo de peso asociado a la última muestra de la capacidad del canal y como por un lado la predicción más estable en cuando a dejar de oscilar después de un cierto tiempo no tiene la atribución de actualizarse cuando la capacidad del canal se ve alterada por algún motivo. Sin embargo usando un peso proporcional contamina la referencia encontrada añadiendo comportamiento propio del módulo utilizado pero se tiene certeza de que bajo este modo se puede actualizar la capacidad del canal cuando el nivel de pérdida cambia. Este tema en si tiene un fin solo de exponer este dilema de elección del tipo de peso, sin embargo en la práctica debe ser raro que se dé un caso en donde la capacidad del canal se vea alterada.

## 5.3 Trabajo futuro

Es posible realizar más proyectos relacionados con esta área y que se pueden derivar del trabajo presentado. Se encontró la capacidad del canal y ésta es indiferente al protocolo de evasión de congestión utilizado por lo tanto es posible crear otros módulos con algún otro tipo de controlador, no necesariamente un PI, que también se mantenga en la capacidad del canal. Además si se pudiera lograr mantener la ventana de congestión invariante por parte de los módulos de congestión, dicho protocolo efectivamente lograría converger a la ventana predicha.

Otra área donde sería posible utilizar la capacidad del canal corresponde a la modificación de protocolos ya existentes agregándole de alguna forma inteligencia, por ejemplo la inclusión de límites superiores e inferiores para la ventana de congestión sabiendo con antelación las limitaciones del canal.



# CAPÍTULO 6: BIBLIOGRAFÍA

1. Arianfar, S. TCP's Congestion Control Implementation in Linux Kernel. In *Proceedings of Seminar on Network Protocols in Operating Systems* (p. 16).
2. Bonald, T., & Massoulié, L. (2001, June). Impact of fairness on Internet performance. In *ACM SIGMETRICS Performance Evaluation Review* (Vol. 29, No. 1, pp. 82-91). ACM.
3. Brakmo, L. S., & Peterson, L. L. (1995). TCP Vegas: End to end congestion avoidance on a global Internet. *Selected Areas in Communications, IEEE Journal on*, 13(8), 1465-1480.
4. Chan, Y. C., Lin, C. L., Chan, C. T., & Ho, C. Y. (2006, February). Improving performance of TCP Vegas for high bandwidth-delay product networks. In *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference* (Vol. 1, pp. 6-pp). IEEE.
5. Chuh, Y., Kim, J., Song, Y., & Park, D. (2005, July). F-TCP: light-weight TCP for file transfer in high bandwidth-delay product networks. In *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on* (Vol. 1, pp. 502-508). IEEE.
6. Day, J. D., & Zimmermann, H. (1983). The OSI reference model. *Proceedings of the IEEE*, 71(12), 1334-1340.
7. Fu, Q. (2009, June). Improving throughput in high bandwidth-delay product networks with random packet losses. In *Communications, 2009. ICC'09. IEEE International Conference on* (pp. 1-6). IEEE.
8. Ha, S., Rhee, I., & Xu, L. (2008). CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5), 64-74.
9. Hong, Y., & Yang, O. W. (2010, May). Robust Explicit Congestion Controller Design For High Bandwidth-Delay Product Network:  $AH_{\infty}$  Approach. In *Communications (ICC), 2010 IEEE International Conference on* (pp. 1-5). IEEE.
10. Hung, W. C., & Law, K. L. E. (2008, May). Simple slow-start and a fair congestion avoidance for TCP communications. In *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on* (pp. 001771-001774). IEEE.
11. Jung, H., Kim, S. G., Yeom, H. Y., Kang, S., & Libman, L. (2011, April). Adaptive delay-based congestion control for high bandwidth-delay product networks. In *INFOCOM, 2011 Proceedings IEEE* (pp. 2885-2893). IEEE.
12. Kim, D., Cano, J. C., Manzoni, P., & Toh, C. K. (2006, September). A comparison of the performance of TCP-Reno and TCP-Vegas over MANETs. In *Wireless Communication Systems, 2006. ISWCS'06. 3rd International Symposium on* (pp. 495-499). IEEE.
13. Lakshman, T. V., & Madhow, U. (1997). The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *Networking, IEEE/ACM Transactions on*, 5(3), 336-350.
14. Leith, D., & Shorten, R. (2004, February). H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of PFLDnet* (Vol. 2004).

15. Li, C. L., Siew, J., & Siew, C. K. (2009, December). Transmission protocol for high bandwidth-delay product networks. In *Information, Communications and Signal Processing, 2009. ICICS 2009. 7th International Conference on* (pp. 1-5). IEEE.
16. Long, C., Chai, X., Guan, X., & Zhang, Q. (2009, November). General congestion control for high bandwidth-delay product networks. In *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE* (pp. 1-6). IEEE.
17. Low, S. H., Paganini, F., & Doyle, J. C. (2002). Internet congestion control. *Control Systems, IEEE*, 22(1), 28-43.
18. Ogata, K. (2003). *Ingeniería de control moderna 4ED*. Pearson Educación, capítulo 1-1 y 5-3.
19. Padhye, J., Firoiu, V., Towsley, D., & Kurose, J. (1998, October). Modeling TCP throughput: A simple model and its empirical validation. In *ACM SIGCOMM Computer Communication Review* (Vol. 28, No. 4, pp. 303-314). ACM.
20. Wang, J., Gong, H., & Chen, J. (2007, August). C3P: a cooperant congestion control protocol in high bandwidth-delay product networks. In *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on* (pp. 687-692). IEEE.
21. Yang, Y. R., Chan, C. T., Wang, P. C., & Chen, Y. C. (2005). Performance enhancement of XCP for high bandwidth-delay product networks. In *Advanced Communication Technology, 2005, ICACT 2005. The 7th International Conference on* (Vol. 1, pp. 456-461). IEEE.
22. Chan, Y. C., Lin, C. L., Chan, C. T., & Ho, C. Y. (2006, February). Improving performance of TCP Vegas for high bandwidth-delay product networks. In *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference* (Vol. 1, pp. 6-pp). IEEE.

# ANEXOS

## Código tcp\_conn\_alt en OPNET para simulaciones

### Desde Linea 973

```
if (!((SACK_PERMITTED) && (tcp_seq_gt (scoreboard_ptr->recovery_end,
seg_ACK))))
{
    /* Store the value of congestion window. */
    cwnd_old = cwnd;

    /* Perform increment based on the fact that cwnd
    /* should increase by 1 MSS per round trip time. */

    alpha = (double) ((double) (cwnd_pred - cwnd)/ (double)
cwnd_pred);

    if (alpha < 0){
        alpha = 0;
    }

    // printf("alpha = %f\n", alpha);

    cwnd += (int) ( (double) (snd_mss * snd_mss * alpha/ cwnd) );

    // printf("current sequence number: %d\n", snd_una);

    /* Perform increment based on the fact that cwnd
    /* should increase by 1 MSS per round trip time.
    /* cwnd += snd_mss * snd_mss / cwnd;

    /* Make sure that congestion window was increased. It might happen that,
    /*
    /* if the congestion window is sufficiently large, and the integer
    /*
    /* arithmetic is used, the above formula will cease to increase cwnd.
    /*
    /* RFC 2581 (TCP Congestion Control) states that in that case the cwnd
    /*
    /* be incremented by 1 byte. Thus if we did not increase the congestion
    /*
    /* window using the equation above, increase it by 1 byte.
    /*
    /* if (cwnd_old == cwnd)
    {
    cwnd += 1; comentamos el incremento de cwnd cuando cwnd_old == cwnd.
    } */
}
```

```
}
```

## Desde línea 2066

```
/* drop the slow start threshold to half the current */
/* flight size */

// beta = (3 - alpha)/(3 + alpha);
beta = cwnd_pred/cwnd;

if (beta > 1){
    beta = 1;
}

ssthresh = (double) ((snd_max - snd_una) * beta);
//ssthresh = (int) (( (double) snd_max - (double) snd_una) *
beta);
// printf("-----\n");
// printf("ssthresh: %d\n", ssthresh);
// printf("beta: %f\n", beta);
// printf("ssthresh observed: %d\n", ((double) (snd_max - snd_una)) * beta
);

/* drop the slow start threshold to half the current */
/* flight size (RFC 2581, eq. 3)
*/
// ssthresh = (snd_max - snd_una)/2;

delta = snd_una - old_seq_una;
old_seq_una = snd_una;
// cwnd_pred = snd_mss*pow(8*delta/(3*snd_mss),0.5);

cwnd_pred = (loss_counter * cwnd_pred +
snd_mss*pow(8*delta/(3*snd_mss),0.5))/(loss_counter + 1);
printf("alpha: %f, beta: %f, cwnd_pred: %d, cwnd: %d, snd_una: %d,
old_una: %d, mss: %d, delta: %d\n", alpha, beta, cwnd_pred, cwnd, snd_una,
old_seq_una, snd_mss, delta);

cwndmax = (loss_counter * cwndmax + cwnd)/(loss_counter + 1);
// cwnd_cumulative += cwnd;
// printf("cwnd: %d, cwndmax: %f\n", cwnd, cwndmax);
loss_counter++;
```

# Código tcp\_conn\_pi en OPNET para simulaciones

## Desde línea 951

```
if (cwnd < ssthresh)
{
/* If we're still doing slow-start, then open the          */
/* window exponentially. Each time an ACK is received,    */
/* increase the congestion window by one. Note that      */
/* with this approach the growth in window size may not  */
/* be exactly exponential because the receiver may      */
/* delay sending ACKs (i.e., it may send a single ACK   */
/* for more than one data pACKet received. In other    */
/* words, the number of ACKnowledged bytes may be more  */
/* than one MSS.) Refer RFC-2001 for more details.      */
/*
cwnd += snd_mss;

intZ_switch = 0; // reset integrador estevez salas
//printf("time: %4.15f - reset integrador\n", op_sim_time() );

/* Limit the value of the congestion window to the slow  */
/* start threshold value.                                */
cwnd = (cwnd < ssthresh) ? cwnd : ssthresh;
}
else
{
/* Otherwise perform congestion avoidance increment      */
/* by-one. However, do not start congestion avoidance,   */
/* if we fast recovery has not been completed for SACK. */
}

if (!(SACK_PERMITTED) && (tcp_seq_gt (scoreboard_ptr->recovery_end,
seg_ACK)))
{
/* Store the value of congestion window.                */
cwnd_old = cwnd;
// estevez salas
/* Perform increment based on the fact that cwnd      */
/* should increase by 1 MSS per round trip time.      */

delta = snd_una - old_seq_una;

//cwnd_pred = (loss_counter * cwnd_pred +
snd_mss*pow(8*delta/(3*snd_mss),0.5))/(loss_counter + 1);

// cada X cantidad de veces grabar para aumentar la velocidad de simulacion
***** !!!!!!!!!!!!!!!
// op_stat_scalar_write ("cwnd_pred", (double) cwnd_pred);
// op_stat_scalar_write ("time", (double) op_sim_time());

//if (op_sim_time() < 1010) {
// Kn1 = 1/cwnd_pred; // 70000;
//} else {

Kn1 = (double) 1/10000;

//}

Kn2 = Kn1/1000;
```

```

if (intZ_switch){
intZ = cwnd_pred - cwnd + intZ;
    } else {
cwnd = cwnd_pred;
intZ = 0;
intZ_switch = 1;
    }

//printf("Integrator value: %4.15f\n", intZ);

alpha = (double) ((double) (cwnd_pred - cwnd) * Kn1) + Kn2 * intZ * alpha;

    //beta_p = (3-alpha)/(3+alpha);

    //op_stat_scalar_write ("fair", beta_p >= beta);

// printf("time: %1.14f      alpha = %1.14f, Kn1 = %1.14f\n", op_sim_time(),
alpha, Kn1);

    if (alpha < 0){
alpha = 0;
        }

// printf("alpha = %f\n", alpha);

cwnd += (int) ( (double) (snd_mss * snd_mss * alpha/ cwnd) );
op_stat_scalar_write ("ssthresh", (double) ssthresh);

// printf("current sequence number: %d\n", snd_una);

/* Perform increment based on the fact that cwnd      */
/* should increase by 1 MSS per round trip time.      */
/* cwnd += snd_mss * snd_mss / cwnd;

cwnd += 1; comentamos el incremento de cwnd cuando cwnd_old == cwnd.
    } */

```

## Desde la línea 2112

```
/* drop the slow start threshold to half the current      */
/* flight size                                           */

// beta = (3 - alpha)/(3 + alpha); // based on TDA
// beta = (double) cwnd_pred/cwnd;

beta_p = (3-alpha)/(3+alpha);
beta = min( (double) (3-alpha)/(3+alpha) , (double) cwnd_pred/cwnd );

//alpha_p = (double) 3 * (1-beta)/(1+beta);

if (beta > 1){
    beta = 1;
}

op_stat_scalar_write ("fair", beta_p >= beta);

ssthresh = (double) ((snd_max - snd_una) * beta);
//ssthresh = (int) (( (double) snd_max - (double) snd_una) *
beta);
// printf("-----\n");
// printf("ssthresh: %d\n", ssthresh);
// printf("beta: %f\n", beta);
// printf("ssthresh observed: %d\n", ((double) (snd_max - snd_una)) * beta
);

/* drop the slow start threshold to half the current      */
/* flight size (RFC 2581, eq. 3)                          */
*/
// ssthresh = (snd_max - snd_una)/2;

delta = snd_una - old_seq_una;
old_seq_una = snd_una;
// cwnd_pred = snd_mss*pow(8*delta/(3*snd_mss),0.5);

cwnd_pred = (loss_counter * cwnd_pred +
snd_mss*pow(8*delta/(3*snd_mss),0.5))/(loss_counter + 1);

// cwndmax = (loss_counter * cwndmax + cwnd)/(loss_counter + 1);
// cwnd_cumulative += cwnd;
// printf("cwnd: %d, cwndmax: %f\n", cwnd, cwndmax);

loss_counter++;

//op_stat_scalar_write ("cwndmax", (double) cwndmax);
op_stat_scalar_write ("cwnd_pred", (double) cwnd_pred);
op_stat_scalar_write ("time", (double) op_sim_time());

// Statistics

mean_cwnd = (stat_counter * mean_cwnd + cwnd)/(stat_counter + 1);
var_cwnd = (stat_counter * var_cwnd + (cwnd - mean_cwnd)*(cwnd -
mean_cwnd))/(stat_counter + 1);
stat_counter++;
```

## Código Tcp\_Reno utilizado con inclusión del cálculo de la capacidad del canal

```
#include <linux/module.h>
#include <net/tcp.h>
#include <linux/time.h>
//#include "ns-linux-c.h"
//#include "ns-linux-util.h"

/*initialize variables */

static struct tcp_naive_reno{
    u32 loss_ctr;
    int delta;
    int old_seq_una;
    int order2;
}datos;

static u32 sqrt_owl(u32 N_owl){

int i_owl;
int x_owl;
int y_owl;

if(N_owl<0)
printk(KERN_INFO "Raiz cuadrada negativa!!!");
if(N_owl==0)
    return 0;
else{
    x_owl=0x0fffffff;
    for(i_owl=0;i_owl<32;++i_owl){

        y_owl=(x_owl+(N_owl/x_owl))/2;
        //printk(KERN_INFO "%u, %u\n" , y_owl, x_owl);
        if(y_owl>=x_owl)
            return x_owl;
        x_owl=y_owl;
    }
    printk(KERN_INFO "Error en el sqrt_owl!!!\n");
    return x_owl;
}
}

static void tcp_naive_reno_init(struct sock *sk)
{

    struct tcp_sock *tp = tcp_sk(sk);

    tp->snd_cwnd_cnt = 0;

    /* Ensure the MD arithmetic works. This is somewhat pedantic,
     * since I don't think we will see a cwnd this large. :) */
```



```

        tp->snd_cwnd_clamp = min_t(u32, tp->snd_cwnd_clamp,
0xffffffff/128);

        printk(KERN_INFO "0 _____NUEVA PRUEBA NAIVE RENO_____");
    }

    /* opencwnd */
    static void tcp_naive_reno_cong_avoid(struct tcp_sock *tp, u32 ACK,
u32 rtt, u32 in_flight, int flag)
    {
        if (tp->snd_cwnd < tp->snd_ssthresh) {
            tp->snd_cwnd++;

            //printk(KERN_INFO "1 Naive Slow_Start");

        } else {
            //contador de paquetes
            /*if (tp->snd_cwnd_cnt >= tp->snd_cwnd) {
                if (tp->snd_cwnd < tp->snd_cwnd_clamp)
                    tp->snd_cwnd++;
                tp->snd_cwnd_cnt = 0;*/

            //contador de bytes
            if (tp->bytes_acked >= tp->snd_cwnd*tp->mss_cache) {
                tp->bytes_acked -= tp->snd_cwnd*tp-
>mss_cache;
                if (tp->snd_cwnd < tp->snd_cwnd_clamp)
                    tp->snd_cwnd++;

            } else {
                tp->snd_cwnd_cnt++;
            }
            //printk(KERN_INFO "2 Naive increase cwnd");
        }
    }

    /* ssthreshold should be half of the congestion window after a loss */
    static u32 tcp_naive_reno_ssthresh(struct tcp_sock *tp)
    {
        //printk(KERN_INFO "3 Naive decrease cwnd");

        datos.delta=tp->snd_una-datos.old_seq_una;
        datos.old_seq_una=tp->snd_una;

        tp->snd_cwnd_pred=(datos.loss_ctr*tp-
>snd_cwnd_pred+sqrt_owl(8*datos.delta/(3*tp->mss_cache)))/(datos.loss_ctr+1)
+ datos.loss_ctr%2; //en paquetes
        printk(KERN_INFO "%d", datos.loss_ctr%2); //
sqrt_owl(8*datos.delta/(3*tp->mss_cache)) );
        datos.loss_ctr++;

        return max(tp->snd_cwnd >> 1U, 2U);
    }

```

```

}

/* congestion window should be equal to the slow start threshold
(after slow start threshold set to half of cwnd before loss). */
static u32 tcp_naive_reno_min_cwnd(struct tcp_sock *tp)
{
    return tp->snd_ssthresh;
}

static struct tcp_congestion_ops tcp_naive_reno = {
    .init      = tcp_naive_reno_init,
    .ssthresh  = tcp_naive_reno_ssthresh,
    .cong_avoid = tcp_naive_reno_cong_avoid,
    .min_cwnd  = tcp_naive_reno_min_cwnd,

    .owner     = THIS_MODULE,
    .name      = "naive_reno"
};

static int __init naive_reno_register(void)
{
    // BUILD_BUG_ON(sizeof(struct naive_reno) > ICSK_CA_PRIV_SIZE);
    return tcp_register_congestion_control(&tcp_naive_reno);
}

static void __exit naive_reno_unregister(void)
{
    tcp_unregister_congestion_control(&tcp_naive_reno);
}

module_init(naive_reno_register);
module_exit(naive_reno_unregister);

MODULE_AUTHOR("Claudio Estevez");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("naive_reno");

```

## Código del módulo tcp\_conn\_alt en Linux

```
#include <linux/module.h>
#include <net/tcp.h>
#include <linux/time.h>

u32 give_me_seq(struct sock*, int*);
static struct conn_alt {
    u32 ai;
    u32 seq_prev;
    u32 alpha_last;
    int no_seq;
    int limit;
    int AI_value;
    int beta;
    u32 loss_ctr;
    int no_loss;
    int n_alpha;           //normalización de alpha
    int n_beta;           //normalización de beta
    int w;                 //weight para el calculo de cwnd_pred
    int cwnd_pred;        //prediction
    int delta;
    int old_seq_una;
    int order2;
}datos;

static u32 sqrt_owl(u32 N_owl){

    int i_owl;
    int x_owl;
    int y_owl;

    if(N_owl<0)
        printk(KERN_INFO "Raiz cuadrada negativa!!!");
    if(N_owl==0)
        return 0;
    else{
        x_owl=0xffffffff;
        for(i_owl=0;i_owl<32;++i_owl){

            y_owl=(x_owl+(N_owl/x_owl))/2;
            if(y_owl>=x_owl)
                return x_owl;
            x_owl=y_owl;
        }
        printk(KERN_INFO "Error en el sqrt_owl!!!\n");
        return x_owl;
    }
}

static void conn_alt_init(struct sock *sk)
{
```

```

struct tcp_sock *tp = tcp_sk(sk);
struct conn_alt *ca = inet_csk_ca(sk);

ca->ai = 0;
datos.no_seq=0;
datos.seq_prev=0;
datos.alpha_last=0;
datos.order2=1024; // 2^10
datos.limit=0xffffffff;
datos.loss_ctr=0;
datos.no_loss=1;
datos.w = 1;
datos.n_alpha=1024;//0x00100000;
datos.n_beta=1024;//0x00100000;
datos.cwnd_pred=1;
datos.old_seq_una=0;

/* Ensure the MD arithmetic works. This is somewhat pedantic,
 * since I don't think we will see a cwnd this large. :) */
tp->snd_cwnd_clamp = min_t(u32, tp->snd_cwnd_clamp,
0xffffffff/128);

tp->snd_wnd = 0xffffffff;
printk(KERN_INFO "_____NUEVA PRUEBA CONN ALT_____\n");
}

static void conn_alt_cong_avoid(struct sock *sk, u32 adk, u32
in_flight)
{
struct tcp_sock *tp = tcp_sk(sk);

int sysctl_tcp_abc=1;//hardcoding
if (!tcp_is_cwnd_limited(sk, in_flight))
return;

/* In "safe" area, increase. */
if(tp->snd_cwnd < tp->snd_ssthresh){ //FS: altered
datos.ssthresh2
//tcp_slow_start(tp);
tp->snd_cwnd = tp->snd_ssthresh;
/*tp->snd_cwnd = datos.cwnd_pred;
if (tp->snd_cwnd < tp->snd_cwnd_clamp)
tp->snd_cwnd++;*/
}

/* In dangerous area, increase slowly. */
else if (sysctl_tcp_abc) {//hardcoding
/* RFC3465: Appropriate Byte Count
 * increase once for each full cwnd ACKed
 */

if (datos.no_loss){
// CE: asegurarse que tcp_abc = 1 (enabled) en
opciones_config

```

```

        if (tp->bytes_ACKed >= tp->snd_cwnd*tp->mss_cache){
            tp->bytes_ACKed -= tp->snd_cwnd*tp-
>mss_cache;
            if (tp->snd_cwnd < tp->snd_cwnd_clamp)
                tp->snd_cwnd++;
        }
    }else{
        if(datos.cwnd_pred > tp->snd_cwnd)
            datos.AI_value = (datos.n_alpha*(datos.cwnd_pred-tp-
>snd_cwnd))/datos.cwnd_pred;
        else
            datos.AI_value=162;

        if(datos.AI_value<=162){ //162 para ser
consistente con el beta maximo 0.9, 3/19*1024
            datos.AI_value=162;

        }

        datos.limit = tp->snd_cwnd*tp-
>mss_cache*datos.n_alpha/datos.AI_value;

        if (tp->bytes_ACKed >= datos.limit ){
            tp->bytes_ACKed -= datos.limit;
if (tp->snd_cwnd < tp->snd_cwnd_clamp)
                tp->snd_cwnd++;

        }

    }
} else {
    printk(KERN_INFO "abc not hardcoded");
    tcp_cong_avoid_ai(tp, tp->snd_cwnd);
}
}

static u32 conn_alt_ssthresh(struct sock *sk)
{
    struct tcp_sock *tp = tcp_sk(sk);
    u32 seq_now,alpha;

    seq_now = give_me_seq(sk,&datos.no_seq);
    if((seq_now>datos.seq_prev) && !datos.no_seq)
        alpha = seq_now - datos.seq_prev;
        datos.seq_prev=seq_now;
        datos.alpha_last=alpha;
    }
    else if(!datos.no_seq){
        alpha = 0xffffffff-datos.seq_prev+seq_now;
        datos.seq_prev=seq_now;
        datos.alpha_last=alpha;
    }
}

```

```

    }else{
        alpha=datos.alpha_last;
        datos.no_seq=0;
    }

    datos.beta = datos.n_beta*(3*datos.n_alpha-
datos.AI_value)/(3*datos.n_alpha+datos.AI_value);

    if(datos.beta>=datos.n_beta)
        datos.beta=9*datos.n_beta/10;

    datos.delta=tp->snd_una-datos.old_seq_una;
    datos.old_seq_una=tp->snd_una;

    datos.cwnd_pred=(
        >mss_cache)) + datos.w*sqrt_owl(8*datos.delta/(3*tp-
        datos.cwnd_pred*(datos.loss_ctr-datos.w+1)
    )/(datos.loss_ctr+1) + datos.loss_ctr%2; //en paquetes
    datos.loss_ctr++;

    tp->snd_cwnd_pred=datos.cwnd_pred;
    datos.no_loss=0;
    return max(tp->snd_cwnd*datos.beta >> 10U, 2U);
}

static struct tcp_congestion_ops tcp_conn_alt = {
    .init = conn_alt_init,
    .ssthresh = conn_alt_ssthresh,
    .cong_avoid = conn_alt_cong_avoid,
    .owner = THIS_MODULE,
    .name = "conn_alt"
};

static int __init conn_alt_register(void)
{
    BUILD_BUG_ON(sizeof(struct conn_alt) > ICSK_CA_PRIV_SIZE);
    return tcp_register_congestion_control(&tcp_conn_alt);
}

static void __exit conn_alt_unregister(void)
{
    tcp_unregister_congestion_control(&tcp_conn_alt);
}

module_init(conn_alt_register);
module_exit(conn_alt_unregister);

u32 give_me_seq(struct sock *sk,int *flag){

```

```
    if(tcp_sk(sk)->high_seq==0){//la probabilidad de que un seq numero
sea cero es casi nula
        *flag=1;
        return 0;
    }
    else{
return tcp_sk(sk)->high_seq;
//return (sk->sk_send_head)->seq;
    }
// return tcp_sk(sk)->high_seq;
}

MODULE_AUTHOR("Claudio Estevez");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("conn_alt");
```

## Código del módulo tcp\_conn\_pi en Linux

```
#include <linux/module.h>
#include <net/tcp.h>
#include <linux/time.h>

u32 give_me_seq(struct sock*, int*);
static struct conn_pi {
    int no_seq;
    int limit;
    int AI_value;
    int beta;
    int no_loss;
    //int weight;
    int cwnd_pred;      //prediction
    int delta;
    int old_seq_una;
    //int order2;
    int Zswitch;      //variables controlador PI
    int K1;
    int K2;
    int intZ;
    u32 ai;
    u32 seq_prev;
    u32 alpha_last;
    u32 loss_ctr;
}datos;
static struct conn_pi_n {
    int alpha;      //normalización de alpha
    int beta;      //normalización de beta
    int K1;
    int K2;
}n;

static u32 sqrt_owl(u32 N_owl){

    int i_owl;
    int x_owl;
    int y_owl;

    if(N_owl<0)
    printk(KERN_INFO "Raiz cuadrada negativa!!!");
    if(N_owl==0)
        return 0;
    else{
        x_owl=0x0fffffff;
        for(i_owl=0;i_owl<32;++i_owl){

            y_owl=(x_owl+(N_owl/x_owl))/2;

            if(y_owl>=x_owl)
                return x_owl;
            x_owl=y_owl;
        }
    }
}
```



```

    printk(KERN_INFO "Error en el sqrt_owl!!!\n");
    return x_owl;
}
}

static void conn_pi_init(struct sock *sk)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct conn_pi *ca = inet_csk_ca(sk);

    ca->ai = 0;
    datos.no_seq=0;
    datos.seq_prev=0;
    datos.alpha_last=0;
    datos.limit=0xffffffff;
    datos.loss_ctr=0;
    datos.no_loss=1;
    n.alpha=1024;//0x00100000;
    n.beta=1024;//0x00100000;
    datos.cwnd_pred=1;
    datos.old_seq_una=0;
    datos.Zswitch=0;
    datos.intZ=0;
    datos.K1=1;
    datos.K2=1;
    n.K1=10000;
    n.K2=10000000;
    /* Ensure the MD arithmetic works. This is somewhat pedantic,
     * since I don't think we will see a cwnd this large. :) */
    tp->snd_cwnd_clamp = min_t(u32, tp->snd_cwnd_clamp,
0xffffffff/128);

    tp->snd_wnd = 0xffffffff;
    printk(KERN_INFO "_____NUEVA PRUEBA CONN ALT_____\n");
}

static void conn_pi_cong_avoid(struct sock *sk, u32 adk, u32
in_flight)
{
    struct tcp_sock *tp = tcp_sk(sk);

    int sysctl_tcp_abc=1;//hardcoding

    if (!tcp_is_cwnd_limited(sk, in_flight))
        return;

    /* In "safe" area, increase. */
    if(tp->snd_cwnd < tp->snd_ssthresh){

    tp->snd_cwnd = tp->snd_ssthresh;
    datos.Zswitch=0;
}
}

```

```

/* In dangerous area, increase slowly. */
else if (sysctl_tcp_abc) { //hardcoding
    /* RFC3465: Appropriate Byte Count
    * increase once for each full cwnd ACKed
    */

if (datos.no_loss){
    // CE: asegurarse que tcp abc = 1 (enabled) en
opciones_config

    if (tp->bytes_ACKed >= tp->snd_cwnd*tp->mss_cache){
        tp->bytes_ACKed -= tp->snd_cwnd*tp-
>mss_cache;
        if (tp->snd_cwnd < tp->snd_cwnd_clamp)
            tp->snd_cwnd++;
    }
} else {

    if(datos.Zswitch){
        datos.intZ=datos.cwnd_pred-tp->snd_cwnd + datos.intZ;
    }
    else{
        tp->snd_cwnd=datos.cwnd_pred;
        datos.intZ=0;
        datos.Zswitch=1;
    }

    if(datos.cwnd_pred > tp->snd_cwnd)
        datos.AI_value = n.alpha*(datos.K1*(datos.cwnd_pred-tp-
>snd_cwnd)/n.K1 + datos.K2*datos.intZ/n.K2);

    else
        datos.AI_value=162;

    if(datos.AI_value<=162){ //162 para ser
consistente con el beta maximo 0.9, 3/19*1024
        datos.AI_value=162;
    }

    datos.limit = tp->snd_cwnd*tp-
>mss_cache*n.alpha/datos.AI_value;
    if (tp->bytes_ACKed >= datos.limit ){
        tp->bytes_ACKed -= datos.limit;
        if (tp->snd_cwnd < tp->snd_cwnd_clamp)
            tp->snd_cwnd++;
    }
}
} else {

```

```

        printk(KERN_INFO "abc not hardcoded");
        tcp_cong_avoid_ai(tp, tp->snd_cwnd);
    }

}

static u32 conn_pi_ssthresh(struct sock *sk)
{
    struct tcp_sock *tp = tcp_sk(sk);
    u32 seq_now,alpha;

    seq_now = give_me_seq(sk,&datos.no_seq);
    if((seq_now>datos.seq_prev) && !datos.no_seq){
alpha = seq_now - datos.seq_prev;
        datos.seq_prev=seq_now;
        datos.alpha_last=alpha;
    }
    else if(!datos.no_seq){
        alpha = 0xffffffff-datos.seq_prev+seq_now;
        datos.seq_prev=seq_now;
        datos.alpha_last=alpha;

    }else{
        alpha=datos.alpha_last;
        datos.no_seq=0;

    }

    datos.beta = n.beta*(3*n.alpha-
datos.AI_value)/(3*n.alpha+datos.AI_value);

    if(datos.beta>=n.beta)
        datos.beta=9*n.beta/10;

    datos.delta=tp->snd_una-datos.old_seq_una;
    datos.old_seq_una=tp->snd_una;

    datos.cwnd_pred=( sqrt_owl(8*datos.delta/(3*tp->mss_cache)) +
datos.cwnd_pred*datos.loss_ctr )/(datos.loss_ctr+1) + datos.loss_ctr%2;
    //en paquetes
    datos.loss_ctr++;

    tp->snd_cwnd_pred=datos.cwnd_pred;
    datos.no_loss=0;

    return min(max(tp->snd_cwnd*datos.beta >> 10U,
2U),datos.cwnd_pred);
}

```

```

static struct tcp_congestion_ops tcp_conn_pi = {
    .init          = conn_pi_init,
    .ssthresh     = conn_pi_ssthresh,
    .cong_avoid   = conn_pi_cong_avoid,

    .owner        = THIS_MODULE,
    .name         = "conn_pi"
};

static int __init conn_pi_register(void)
{
    BUILD_BUG_ON(sizeof(struct conn_pi) > ICSK_CA_PRIV_SIZE);
    return tcp_register_congestion_control(&tcp_conn_pi);
}

static void __exit conn_pi_unregister(void)
{
    tcp_unregister_congestion_control(&tcp_conn_pi);
}

module_init(conn_pi_register);
module_exit(conn_pi_unregister);

u32 give_me_seq(struct sock *sk, int *flag){
    if(tcp_sk(sk)->high_seq==0){
        *flag=1;
        return 0;
    }
    else{
        return tcp_sk(sk)->high_seq;
        //return (sk->sk_send_head)->seq;
    }
    // return tcp_sk(sk)->high_seq;
}

MODULE_AUTHOR("Claudio Estevez");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("conn_pi");

```

## Código del bash opciones\_buff

```
#!/bin/sh
sudo sysctl -w net.ipv4.tcp_abc=1
sudo sysctl -w net.ipv4.tcp_adv_win_scale=7
#default 2
sudo sysctl -w net.ipv4.tcp_abort_on_overflow=1
#sudo sysctl -w net.ipv4.tcp_allowed_congestion_control=reno
sudo sysctl -w net.ipv4.tcp_app_win=31
sudo sysctl -w net.ipv4.tcp_base_mss=512
#sudo sysctl -w net.ipv4.tcp_bic NO EXISTE
#sudo sysctl -w net.ipv4.tcp_bic_low_window NO EXISTE
#sudo sysctl -w net.ipv4.tcp_bic_fast_convergence NO EXISTE
sudo sysctl -w net.ipv4.tcp_dma_copybreak=4096
sudo sysctl -w net.ipv4.tcp_dsACK=1

sudo sysctl -w net.ipv4.tcp_ecn=0
#default 2
sudo sysctl -w net.ipv4.tcp_fACK=0
#default 1
sudo sysctl -w net.ipv4.tcp_fin_time-out=60
sudo sysctl -w net.ipv4.tcp_frto=0
#default 2
sudo sysctl -w net.ipv4.tcp_frto_response=0
#no se puede desactivar
sudo sysctl -w net.ipv4.tcp_keepalive_intvl=75
sudo sysctl -w net.ipv4.tcp_keepalive_probes=9
sudo sysctl -w net.ipv4.tcp_keepalive_time=7200
sudo sysctl -w net.ipv4.tcp_low_latency=0
sudo sysctl -w net.ipv4.tcp_max_orphans=8192
#default 8192
sudo sysctl -w net.ipv4.tcp_max_syn_bACKlog=1024
#default=1024
sudo sysctl -w net.ipv4.tcp_max_tw_buckets=180000
#default= 180000
sudo sysctl -w net.ipv4.tcp_moderate_rcvbuf=1

#default=1 si està en 0 iperf no obtiene data

#default 46560 62080 93120
sudo sysctl -w net.ipv4.tcp_mtu_probing=0
#default=0
sudo sysctl -w net.ipv4.tcp_no_metrics_save=1
#default=0 1 para hacer pruebas
sudo sysctl -w net.ipv4.tcp_orphan_retries=0
#default sistema 0, default papel 8
sudo sysctl -w net.ipv4.tcp_reordering=3
#default 3
sudo sysctl -w net.ipv4.tcp_retrans_collapse=0
#default 1
sudo sysctl -w net.ipv4.tcp_retries1=3
#default 3
sudo sysctl -w net.ipv4.tcp_retries2=15
#default 15
```

```

sudo sysctl -w net.ipv4.tcp_rfc1337=0
#default 0
sudo sysctl -w net.ipv4.tcp_sACK=1
#default 1
sudo sysctl -w net.ipv4.tcp_slow_start_after_idle=1

#default 1
sudo sysctl -w net.ipv4.tcp_stdurg=0
#default 0
sudo sysctl -w net.ipv4.tcp_syn_retries=5
#default 5
sudo sysctl -w net.ipv4.tcp_synACK_retries=5
#default 5
sudo sysctl -w net.ipv4.tcp_syncookies=0
#default =1
sudo sysctl -w net.ipv4.tcp_timestamps=1
#default 1
sudo sysctl -w net.ipv4.tcp_tso_win_divisor=3
#default 3
sudo sysctl -w net.ipv4.tcp_tw_recycle=0
#default 0
sudo sysctl -w net.ipv4.tcp_tw_reuse=0
#default 0
#sudo sysctl -w net.ipv4.tcp_vegas_cong_avoid NO EXISTE
#sudo sysctl -w net.ipv4.tcp_westwood NO EXISTE
sudo sysctl -w net.ipv4.tcp_window_scaling=1
#default 1

sudo sysctl -w net.ipv4.tcp_workaround_signed_windows=0
#default 0
#sudo sysctl -w net.ipv4.tcp_
#sudo sysctl -w net.ipv4.tcp_

#Relacionado con buffers
sudo sysctl -w net.ipv4.tcp_mem='1048575 33554431 1073741823'
#sudo sysctl -w net.ipv4.tcp_mem='1073741823 1073741823
1073741823'
#sudo sysctl -w net.ipv4.tcp_mem='4096 87380 524288'
#sudo sysctl -w net.ipv4.tcp_mem='2048 43690 262144'

sudo sysctl -w net.ipv4.tcp_wmem='1048575 33554431 1073741823'
#sudo sysctl -w net.ipv4.tcp_wmem='1661992959 1661992959 1661992959'
#sudo sysctl -w net.ipv4.tcp_wmem='4096 87380 524288'
#sudo sysctl -w net.ipv4.tcp_wmem='2048 43690 262144'

sudo sysctl -w net.ipv4.tcp_rmem='1048575 33554431 1073741823'
#sudo sysctl -w net.ipv4.tcp_wmem='1073741823 1073741823
1073741823'
#sudo sysctl -w net.ipv4.tcp_wmem='1661992959 1661992959
1661992959'
#sudo sysctl -w net.ipv4.tcp_rmem='4096 87380 524288'
#sudo sysctl -w net.ipv4.tcp_rmem='2048 43690 262144'

#sudo sysctl -w net.ipv4.tcp_rmem='268435455 268435455 268435455'
#sudo sysctl -w net.ipv4.tcp_rmem='1661992959 1661992959
1661992959'

```

```
#sudo sysctl -w net.ipv4.tcp_rmem='1073741823 1073741823
1073741823'
#sudo sysctl -w net.ipv4.tcp_rmem = '4096 87380 8388608'

sudo sysctl -w net.core.optmem_max=10240
sudo sysctl -w net.core.wmem_default=114688
sudo sysctl -w net.core.wmem_max=131071
sudo sysctl -w net.core.rmem_default=114688
sudo sysctl -w net.core.rmem_max=131071

#a continuacion no listados en papel

sudo sysctl -w net.ipv4.tcp_cookie_size=0
#default 0
sudo sysctl -w net.ipv4.tcp_max_ssthresh=1661992959
#sudo sysctl -w net.ipv4.tcp_max_ssthresh=0 - cambio: dic 5 2011
#default 0
sudo sysctl -w net.ipv4.tcp_thin_dupACK=0
#default 0
sudo sysctl -w net.ipv4.tcp_thin_linear_time-outs=0
#default 0
#sudo sysctl -w net.ipv4.tcp_
```

## Código de script lector de archivos tcp\_probe en MATLAB<sup>®</sup>

```
clc
close all

[fnamepb path] = uigetfile ('/owl/*.info','Open File');

fprintf('Extracting data from file... ');
fileIDpb = fopen([path,fnamepb]);
Cpb = textscan(fileIDpb, '%s', 'delimiter', '\n');
fclose(fileIDpb);
fprintf('done!\n\n');

Lpb = length(Cpb{1});
th = 0.02;
time_pb = zeros(1,Lpb);
cwnd_pb = zeros(1,Lpb);
ssthresh_pb = zeros(1,Lpb);
cwnd_pred = zeros(1,Lpb);

disp('<----- TcpProbe Info ----->')

for n=1:Lpb
    tmp = cell2mat(Cpb{1}(n));
    blnk = strfind(tmp, ' ');
    time_pb(n) = str2double(tmp(1:blnk(1)-1));
    cwnd_pb(n) = str2double(tmp(blnk(6)+1:blnk(7)-1));
    cwnd_pred(n) = str2double(tmp(blnk(8)+1:blnk(9)-1));
    prog=n/Lpb;
    if prog >= th
        th = th + 0.02;
        fprintf('*');
    end
end
disp(' ');

stairs(time_pb,cwnd_pb,'k')
hold on
stairs(time_pb,cwnd_pred,'--b')
axis([min(time_pb),max(time_pb),0,1000])
axis([0,180,0,2.3*mean(cwnd_pred)])
hold off
```