UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL

MARCOS TALAU

NGWA: ESQUEMA DE CONTROLE DE CONGESTIONAMENTO PARA TCP BASEADO NA BANDA DISPONÍVEL

DISSERTAÇÃO

CURITIBA

2012

MARCOS TALAU

NGWA: ESQUEMA DE CONTROLE DE CONGESTIONAMENTO PARA TCP BASEADO NA BANDA DISPONÍVEL

Dissertação apresentada ao Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de "Mestre em Ciências" – Área de Concentração: Telemática.

Orientador: Emilio Carlos Gomes Wille

CURITIBA

2012

Dados Internacionais de Catalogação na Publicação

T137 Talau, Marcos
NGWA: esquema de controle de congestionamento para TCP baseado na banda disponível/ Marcos Talau. – 2012.
84 f. : il. ; 30 cm

Orientador: Emilio Carlos Gomes Wille.
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2012.
Bibliografia: f. 59-63.

1. TCP/IP (Protocolo de rede de computação). 2. Redes de computação – Protocolos. 3. Teoria do controle. 4. Comutação de pacotes (Transmissão de dados). 5. Linux (Sistema operacional de computador). 6. Simulação (Computadores). 7. Engenharia elétrica – Dissertações. I. Wille, Emílio Carlos Gomes, orient. II. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. III. Título.

CDD (22. ed.) 621.3

Biblioteca Central da UTFPR, Câmpus Curitiba

Título da Dissertação Nº 596:

"Esquema de Controle de Congestionamento para TCP Baseado na Banda Disponível".

por

Marcos Talau

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM CIÊNCIAS – Área de Concentração: Telemática, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI – da Universidade Tecnológica Federal do Paraná – UTFPR – Câmpus Curitiba, às 09h30min. do dia 04 de maio de 2012. O trabalho foi aprovado pela Banca Examinadora, composta pelos professores:

Prof. Emilio Carlos Gomes Wille, Dr.
(Presidente)

Prof. Mauro Sergio Pereira Fonseca, Dr.
(PUC)

Profª. Keiko Verônica Ono Fonseca, Dr.
(UTFPR-CT)

Visto da coordenação:

Prof. Fábio Kurt Schneider, Dr.
(Coordenador do CPGEI)

À Anita, Arminda, e Salute.

AGRADECIMENTOS

Deixo aqui um agradecimento a pessoas que de momento me vem a memória, peço desculpas (se for o caso) para o esquecimento de alguém.

Agradecimentos:

Aos Professores Éden Ricardo Dosciati, e Walter Godoy Júnior. O primeiro principalmente pela indicação do programa de mestrado, e ao segundo por me apresentar o Professor Emilio Carlos Gomes Wille, o qual foi orientador deste trabalho.

Ao orientador, Emilio, por sua atenção, dedicação, paciência, e apoio neste trabalho.

Aos colegas do núcleo avançado em tecnologia de comunicações (NATEC) pela amizade e companheirismo; Augusto Foronda, Bozo, Charles Fung, Cubanos, Éden, Kleber Kendy Horikawa Nabas, Marcos Mincov Tenório.

If fifty million people say a foolish thing, it's still a foolish thing. (Bertrand Russell)

RESUMO

TALAU, Marcos. NGWA: ESQUEMA DE CONTROLE DE CONGESTIONAMENTO PARA TCP BASEADO NA BANDA DISPONÍVEL. 84 f. Dissertação – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2012.

O controle de congestionamento padrão do TCP apresenta vários problemas; ele não consegue distinguir se o pacote foi perdido por falha no enlace ou por descarte de pacotes devido a um congestionamento de rede (se a falha foi no enlace não há necessidade de ativar os mecanismos de controle de congestionamento); e o correto ajuste de sua taxa de transmissão requer informação de perdas de pacotes. Neste trabalho é apresentado o *new generalized window advertising* (*NGWA*), que é um novo esquema de controle de congestionamento para o TCP. O *NGWA* traz informações da banda disponível da infraestrutura de rede para os pontos finais da conexão TCP. Seu desempenho foi comparado com TCP New Reno, RED e o TCP padrão via simulações com o *software* NS-3, considerando topologias de rede largamente citadas na literatura. O *NGWA* foi, também, implementado e testado no Linux (versão 2.6.34). O novo método demonstrou ser superior aos comparados, apresentando uma operação mais estável, melhor justiça e menor taxa de perda de pacotes, considerando o elenco de testes realizados.

Palavras-chave: Controle de Congestionamento, TCP, Produto Banda-Atraso, GWA, Linux.

ABSTRACT

TALAU, Marcos. NGWA: AN APPROACH OF TCP CONGESTION CONTROL CONSIDERING THE AVAILABLE BANDWIDTH. 84 f. Dissertação – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2012.

The TCP congestion control mechanism in standard implementations presents several problems; he cannot distinguish if the packet was lost by link failure or by congestion in the net (if the fault was in the link there is no need to active congestion control mechanisms); and the right adjust of your transmission rate requires information from packet loss. This work presents the *new generalized window advertising* (*NGWA*), which is a new congestion control scheme for TCP. The *NGWA* provides information considering the available bandwidth of the network infrastructure to the endpoints of the TCP connection. Results obtained by the *NGWA* approach were compared with those from TCP New Reno, RED, and standard TCP (using the network simulator NS-3), considering network topologies widely cited in the literature. A *NGWA* Linux implementation is also presented. The new method proved to be superior when compared with the traditional approaches, presenting a more stable operation, better fairness and lower packet loss, considering the set of tests carried out.

Keywords: Congestion Control, TCP, Bandwidth-Delay Product, GWA, Linux.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AIMD	*Additive-Increase-Multiplicative-Decrease*
AQM	*Active Queue Management*
CBQ	*Class-Based Queueing*
ECN	*Explicit Congestion Notification*
FIFO	*First In First Out*
GPL	*General Public License*
GWA	*Generalized Window Advertising*
IHL	*Internet Header Length*
MSS	*Maximum Segment Size*
NS	*Network Simulator*
RED	*Random Early Detection*
RTT	*Round-Trip Time*
SACK	*Selective Acknowledgment*
TCP	*Transmission Control Protocol*

SUMÁRIO

1 INTRODUÇÃO

O protocolo de controle de transmissão (TCP) é o mais importante protocolo de transporte para a Internet (JIANG et al., 2009) (WEIGLE; FENG, 2001) (VANGALA; LABRADOR, 2003). Ele é utilizado em uma grande quantidade de redes de computadores pois traz uma forma simples de entrega de dados e contém estratégias de retransmissão e adaptação que são ocultas para as camadas superiores. Isso é suficiente à maior parte das aplicações (MATHIS et al., 2003). Pelas características citadas, o TCP é o protocolo de transporte mais utilizado sobre a Internet (EWALD; KEMP, 2009).

Segundo (COMER, 2000) pode haver perda de pacotes e falhas de transmissões no nível físico de uma rede. Os dados podem ser perdidos, duplicados ou interrompidos sem chegar ao seu destino. O TCP assume que todas as perdas de transmissão são causadas pela ocorrência de um congestionamento, ou seja, se ocorrer uma perda de pacotes devido a um problema no enlace (por exemplo, sem fio) o TCP irá reagir como se isto fosse um congestionamento.

Um congestionamento ocorre quando existem mais dados em uma rede do que ela pode suportar; a solução lógica para resolver este problema corresponde à diminuição da taxa de transmissão de dados (KUROSE; ROSS, 2010). Além disso, o protocolo TCP apresenta outros problemas; ele não consegue distinguir se o pacote foi perdido por falha no enlace ou por descarte de pacotes devido a um congestionamento de rede (se a falha foi no enlace não há necessidade de ativar os mecanismos de controle de congestionamento), e o correto ajuste de sua taxa de transmissão requer informação de perdas de pacotes.

1.1 MOTIVAÇÃO

Muitas propostas para a melhoria do desempenho do protocolo TCP têm sido apresentadas na literatura. Os itens apresentados a seguir serviram de motivação para a proposta apresentada nesta dissertação.

- A banda disponível em um caminho de rede é fundamental para o controle de

congestionamento. Provavelmente o mais importante problema não resolvido no TCP seja o de automaticamente determinar melhor a taxa de transmissão de uma conexão; de posse de informação sobre a banda disponível é possível determinar a taxa mais adequada, não sendo necessário adotar mecanismos de controle, como, por exemplo, o *slow-start* (JAIN; DOVROLIS, 2003).

- A informação de banda disponível pode dar um valor mais apropriado para a variável de tamanho da janela limiar (*ssthresh*), melhorando a fase de *slow-start* do TCP. Quanto mais informações sobre as condições de rede forem disponíveis para o protocolo de transporte, mais eficiente podem ser as transferências de dados (ALLMAN; PAXSON, 1999).

- Com o retorno explícito da banda disponível da rede, o emissor TCP pode controlar o tamanho da janela sem a necessidade da perda de pacotes devido ao congestionamento. A rede deve retornar informações sobre seus recursos para os emissores (*sources*) TCP (GERLA et al., 2000).

- Busca pela redução da complexidade das implementações TCP. Recomendação da retirada do controle de congestionamento do TCP (JIANG et al., 2009).

- A forma mais efetiva de detecção de congestionamento pode ocorrer no *gateway* (FLOYD; JACOBSON, 1993).

- Se o transmissor TCP reconhecer a largura de banda (ter a informação), rapidamente e adequadamente ele pode criar um melhor mecanismo de controle de congestionamento (JIANG et al., 2009) (HASEGAWA; MURATA, 2006).

1.2 OBJETIVOS

Para melhorar o desempenho do controle de congestionamento para o protocolo TCP esta dissertação propõe e analisa um novo esquema, chamado *new generalized window advertising* (*NGWA*), baseado na banda disponível da infraestrutura de rede. Os objetivos específicos da dissertação são:

- Propor e validar um novo esquema de controle de congestionamento para o TCP que seja mais eficiente, tanto na distribuição dos recursos da rede, quanto no controle de congestionamento.

- Utilizando o novo esquema busca-se verificar a possibilidade de remoção de qualquer implementação específica de controle de congestionamento do receptor/transmissor TCP, pois o esquema irá fornecer os limites para transmissão na rede.
- Implementar e testar o esquema descrito em um simulador de rede, e também em um ambiente físico (roteador [1] baseado no sistema operacional GNU/Linux).

1.3 DELIMITAÇÕES DE ESTUDO

Consideram-se, como premissas para o desenvolvimento desta proposta, uma rede cabeada onde o tráfego corresponde a um agregado de fluxos TCP e o roteamento é estático, ou seja, os pacotes IP seguem a mesma rota entre origem e destino. Assume-se, também, que cada roteador presente na rede tem capacidade de informar a quantidade de espaço disponível em suas filas a qualquer instante de tempo.

Com relação à avaliação da proposta considerou-se topologias de rede onde os tempos de viagem (*round-trip times*) entre quaisquer pares origem-destino são fixos e aproximadamente iguais, considerou-se, também, que os dados a serem transmitidos são gerados por fontes do tipo *on/off*.

1.4 ESTRUTURA DA DISSERTAÇÃO

A dissertação foi organizada da seguinte forma: No Capítulo 2 é descrito em detalhes o controle de fluxo e o controle de congestionamento do protocolo de transporte TCP, após isto é apresentada uma série de problemas, finalizando com o levantamento de trabalhos relacionados. O Capítulo 3 apresenta a proposta deste trabalho e resultados obtidos com o simulador de redes (NS-3). No Capítulo 4 é descrita a implementação da proposta no Linux, junto com alguns experimentos. E por fim, o Capítulo 5 revê os principais resultados e apresenta propostas de trabalhos futuros.

[1] A palavra roteador utilizada neste trabalho deve ser interpretada como um *hardware* que realizada roteamento, podendo ser um computador PC ou um roteador tradicional.

2 O PROTOCOLO TCP E SEUS PROBLEMAS

Neste capítulo são descritos os princípios de operação do controle de transmissão do TCP, seguido por detalhes e implementações do controle de congestionamento. Por último são apresentados alguns problemas relacionados ao protocolo TCP, bem como, diversos trabalhos correlatos.

2.1 FUNDAMENTOS

Segundo (COMER, 2000) pode haver perda de pacotes [1] e falhas de transmissões no nível físico de uma rede. Os dados podem ser perdidos, duplicados ou interrompidos sem chegar ao seu destino. O TCP assume que todas as perdas na transmissão são causadas pela ocorrência de um congestionamento, ou seja, se ocorrer uma perda de pacotes devido a um problema de enlace (por exemplo, sem fio) o TCP irá agir como se isto fosse um congestionamento. Na literatura tem-se a definição clássica: um congestionamento é criado quando existem mais dados em uma rede do que ela pode suportar; a solução lógica para resolver este problema é com a diminuição da taxa de transmissão de dados para a rede (TANENBAUM, 1997).

Aproximadamente após dois anos e meio da definição formal do TCP pela RFC 793, foi publicada a primeira RFC (896) relacionada ao controle de congestionamento do protocolo TCP. Nagle verificou que quando o protocolo de transporte TCP era usado juntamente com o protocolo de rede IP, estes estavam sujeitos a problemas causados por interações entre a camada de transporte e de rede, com maior destaque para os *gateways* IP, por serem vulneráveis ao fenômeno de colapso de congestionamento. Posteriormente Jacobson formulou os algoritmos clássicos de controle de congestionamento: *slow-start*, *congestion avoidance*, *fast retransmit* e *fast recovery*, que passaram a fazer parte do padrão de implementação do TCP na RFC 1122 (JACOBSON, 1988) (JACOBSON, 1990).

O protocolo TCP utiliza duas informações para determinar o tamanho da rajada de

[1] Neste trabalho os termos segmento e pacote tem o mesmo significado, porém busca-se o uso adequado quando se estiver dando enfoque ao segmento TCP.

dados que irá transmitir: o campo janela do cabeçalho TCP (W_a) e a janela de congestionamento (W_c), usados, respectivamente, para o controle de fluxo e de congestionamento. O controle de fluxo tem a tarefa de não enviar dados além do que o receptor pode receber; ele consegue fazer isso através da leitura do campo janela do segmento recebido. O transmissor não enviará dados que ultrapassem esta medida; dessa forma evita-se que um lado com um grande poder de transmissão sufoque um receptor que possua poucos recursos. O controle de congestionamento é uma medida que tenta prever a quantidade de dados que a rede irá suportar, e é utilizada pelos dois pontos da conexão. Cada um deles mantém uma janela de congestionamento, baseada em cálculos que tentam prever como está o meio (rede) que os liga. O W_c surgiu no trabalho de (JACOBSON, 1988) como parte do algoritmo *slow-start*.

Implementações TCP utilizam a Equação (1) para determinar a quantidade de dados a ser transmitida (W_t), com isso tenta-se evitar a sobrecarga do receptor e da rede. O W_u é a quantidade de dados já enviados, porém ainda sem confirmação de recebimento (*acknowledgment*).

$$W_t = min[W_a, W_c] - W_u \tag{1}$$

2.2 CONTROLE DE CONGESTIONAMENTO

O controle de congestionamento do TCP surgiu no trabalho de Jacobson (JACOBSON, 1988), baseado no relato de ocorrências de congestionamento na Internet, e investigação de causas. A partir do princípio de conservação de pacotes (um novo pacote não é enviado até que o antigo tenha saído), foram levantados casos onde o princípio pode vir a falhar, propondo-se algoritmos para evitar tais falhas. Estes foram testados no sistema operacional 4BSD (*Berkeley UNIX*). Os algoritmos ali descritos constituem a base do controle de congestionamento da Internet.

Um congestionamento é associado a uma ocorrência de perda de pacotes; é desconsiderada a existência de perdas por falha na rede, pois ela é menor do que um por cento [2] (JACOBSON, 1988) (STEVENS, 1994). A perda de pacotes pode ser indicada por duas formas (STEVENS, 1994) (GERLA et al., 2001):

- Tempo esgotado (*timeout*): Ocorre quando uma confirmação de recebimento de segmento

[2] O controle de congestionamento do TCP baseia-se no princípio de que uma perda foi causada por um congestionamento. Em redes sem fio este método não deveria ser aplicado, pois neste tipo de rede, erros de transmissão e interferências são frequentes.

(ACK) não é recebida em um tempo médio estimado.

- ACK duplicado (*duplicate ACKs*): É definido pela recepção de um ACK com um número de sequência que já havia sido recebido anteriormente. Logo, esse ACK ao chegar ao destino será considerado como duplicado.

Após a detecção de um congestionamento, uma ação deve ser tomada. Nas próximas seções serão descritos os algoritmos *slow-start*, *congestion avoidance*, *fast retransmit* e *fast recovery*, finalizando com a descrição sucinta de algoritmos alternativos (implementações TCP).

2.2.1 ALGORITMOS CLÁSSICOS

Nesta seção são descritos os algoritmos do modelo aumento-aditivo-redução -multiplicativa (AIMD). Os algoritmos do esquema AIMD utilizam a janela de congestionamento (*cwnd* [3]). A utilização dela irá interferir diretamente na quantidade de *bytes* a ser transmitida. A Equação (1) comprova o argumento anterior.

2.2.1.1 SLOW-START

Este algoritmo têm por objetivo ajustar a janela de congestionamento a um valor adequado a situação de congestionamento da rede. No início da conexão ou após uma perda de pacotes por tempo (*timeout*), a janela de congestionamento será igualada a um. Após cada confirmação de recebimento (ACK) a janela é incrementada em um segmento. É considerado que para cada segmento recebido, um ACK de resposta é enviado, desta forma a janela de congestionamento cresce exponencialmente (JACOBSON, 1988).

2.2.1.2 CONGESTION AVOIDANCE

Similar ao *slow-start*, este algoritmo também atua na perda de pacotes por tempo e no recebimento de confirmações. Porém, ao detectar uma perda, a janela de congestionamento é ajustada para a metade do seu valor, e no recebimento de ACKs a janela é incrementada por $1/cwnd$ (JACOBSON, 1988).

Na prática, os algoritmos *slow-start* e *congestion avoidance* devem ser usados em conjunto. Para isso ser possível uma nova variável é utilizada. A variável *ssthresh* indica o ponto de troca dos algoritmos. A Figura 1 descreve a operação de alternância dos algoritmos.

[3] Na literatura e neste trabalho *cwnd* tem o mesmo significado de W_c.

2.2.1.3 FAST RETRANSMIT E FAST RECOVERY

Como seu nome sugere, o *fast retransmit* faz a retransmissão de segmentos considerados perdidos sem aguardar pela expiração de tempo (*timeout*). A sua ativação ocorre por recebimento de um conjunto de ACKs duplicados (esta quantidade é utilizada, pois, o recebimento de segmentos fora de ordem (que gera ACK duplo) em uma quantidade inferior ao conjunto tende a não indicar uma perda. Mas quando o número de ACKs duplicados for igual ou superior a três, a probabilidade de perda do segmento é alta, e é neste momento que o *fast retransmit* irá retransmitir os segmentos considerados perdidos (STEVENS, 1997).

O *fast recovery* é definido pela execução do *fast retransmit* seguida pela execução do algoritmo *congestion avoidance*. Assim como o *slow-start* e o *congestion avoidance* são implementados em conjunto, o *fast retransmit* e o *fast recovery* também são. A Figura 2 contém o pseudocódigo desta implementação.

```
se cwnd < ssthresh
   executar o algoritmo slow-start
senão
   executar o congestion avoidance
```

Figura 1: Pseudocódigo de mudança de algoritmo.

```
// Função executada após a recepção do ACK
// do segmento retransmitido
aguardar_chegada_novo_ack()
   // congestion avoidance
   cwnd = ssthresh

// Principal função do código
recepcao_tres_acks_duplicados()
   ssthresh = cwnd * 0.5
   se sshtresh < dois_segmentos
     ssthresh = 2
   retransmitir_segmento()
   cwnd = ssthresh + 3 * tamanho_segmento
   aguardar_chegada_novo_ack()

// Cada ACK duplicado recebido, executa este código
recebe_dup_ack()
   cwnd = cwnd + tamanho_segmento
```

Figura 2: Implementação conjunta dos algoritmos *fast retransmit* e *fast recovery* (STEVENS, 1997).

2.2.2 IMPLEMENTAÇÕES TCP

As implementações de controle de congestionamento do TCP são algoritmos geralmente compatíveis com o padrão do TCP (*TCP-friendly*). Estes algoritmos podem ser classificados em dois grupos: baseados em perda (*loss-based*) e baseados em atraso (*delay-based*). No primeiro tipo um congestionamento é detectado quando uma perda de pacotes ocorre, e se ajusta a janela de congestionamento para um nível mais baixo. Os algoritmos baseados em atraso fazem uso de valores RTT para detectar um congestionamento. Os dois métodos também podem ser usados em conjunto (KODAMA et al., 2009) (LA; ANANTHARAM, 2000).

Abaixo são descritas algumas das principais implementações TCP existentes.

2.2.2.1 TAHOE, RENO, NEW RENO

A primeira implementação TCP criada para o controle de congestionamento foi o TCP Tahoe. Ele basicamente contém os algoritmos *slow-start*, *congestion avoidance*, e o *fast retransmit* de Jacobson (JACOBSON, 1988), além de refinamentos no cálculo de tempo de viagem (*round-trip time*) (FALL; FLOYD, 1996).

A evolução do Tahoe criou o TCP Reno. Além dos três algoritmos clássicos, ele implementa o *fast recovery* (STEVENS, 1997) (FALL; FLOYD, 1996) (LAI; YAO, 2000). O TCP Reno tem um grande problema: não consegue convergir o tamanho da janela para um valor adequado quando o ambiente de rede é heterogêneo (HASEGAWA; MURATA, 2006).

O TCP New Reno traz melhorias no algoritmo *fast recovery*. Durante o *fast recovery* ao se receber um ACK (com novo número de sequência) que ainda o mantenha na fase de *fast recovery*, o pacote será retransmitido imediatamente. O transmissor irá permanecer nesta fase até que ocorra um *timeout* ou todos os pacotes perdidos sejam retransmitidos com sucesso. O New Reno é útil quando não se tem o recurso de resposta seletiva (SACK) (ZHOU et al., 2007).

2.2.2.2 SACK

A resposta seletiva (SACK) não tem perfil de uma implementação TCP, pois ela é uma opção extra que pode ser adicionada ao protocolo. Quando um segmento é perdido um ACK duplicado é enviado, a recepção deste faz com que todos os segmentos pertencentes a janela sejam retransmitidos; o correto seria enviar apenas o segmento perdido, e não todos os segmentos da janela. O SACK foi criado para resolver este problema. Ao receber pacotes, o

receptor que implemente o SACK irá enviar pacotes ao transmissor informando quais segmentos já foram recebidos (MATHIS et al., 1996).

2.2.2.3 VEGAS

O TCP Vegas tenta antecipar a percepção de um congestionamento pelo monitoramento da diferença entre a velocidade atual de recebimento de pacotes, e a velocidade esperada. A estratégia do Vegas é ajustar a janela de congestionamento para tentar manter um pequeno número de pacotes armazenados nos roteadores ao longo do caminho (LOW et al., 2001). Em testes realizados, o TCP Vegas obteve de 37 a 71% melhor vazão que o Reno (BRAKMO; PETERSON, 1995).

O TCP Vegas apresenta alguns problemas. O TCP Reno utiliza um esquema de controle agressivo que expande a taxa de transmissão (*bandwidth*) até que os pacotes transmitidos sejam perdidos, enquanto que o Vegas é conservador. Quando conexões Reno e o Vegas existirem juntas, as conexões TCP Reno irão ficar com a largura de banda das conexões TCP Vegas (LAI; YAO, 2000). Outro problema está ligado ao RTT. Como o Vegas utiliza valores de RTT para prever um congestionamento, em redes de alta velocidade, o RTT tende a ficar inexpressivo com o aumento da taxa de transmissão, logo, implementações que são baseadas em RTT não irão funcionar adequadamente (HASEGAWA; MURATA, 2006).

2.2.2.4 WESTWOOD

O TCP Westwood (TCPW) faz o controle da janela fim a fim através da estimação contínua da velocidade dos pacotes, pela monitoração da recepção de ACKs. Após uma ocorrência de congestionamento, a estimação obtida é utilizada para computar a janela de congestionamento e o intervalo de troca (*ssthresh*) do *slow-start* . Essa característica torna o TCPW mais robusto a os problemas dos enlaces sem fio. O TCPW obteve melhorias significativas de desempenho em comparação ao Reno e o SACK, principalmente em um ambiente misto (com/sem fio) (GERLA et al., 2001).

2.2.2.5 SYMBIOSIS

A maioria das implementações baseia-se na mudança de parâmetros do modelo AIMD para se adequar a determinado tipo de rede. O TCP Symbiosis é bastante diferente dos demais. Ele é baseado no ajuste do tamanho da janela de uma conexão TCP através de informações físicas e de banda disponível no caminho fim-a-fim da rede. Os algoritmos usados são inspirados

na biofísica. Por análises matemáticas demonstrou-se que o TCP Symbiosis apresenta uma boa escalabilidade sobre as implementações Reno, *HighSpeed*, *Scalable TCP* e *FAST TCP* (HASEGAWA; MURATA, 2006).

2.3 TRABALHOS RELACIONADOS E PROBLEMAS

Os métodos clássicos de controle de congestionamento da Internet são constituídos de mecanismos fim-a-fim juntamente com estratégias de filas nos roteadores. Os procedimentos fim-a-fim são constituídos de algoritmos que tem por objetivo manter uma alta utilização da rede, tentando evitar que a rede fique ociosa (MORRIS, 1997). O método clássico possuí vários problemas:

1. Quando existe um número excessivo de fluxos comunicando-se com um dispositivo o controle de congestionamento não funciona adequadamente (MORRIS, 1997).

2. Ausência de repasse de informação da rede para os emissores TCP (ALLMAN; PAXSON, 1999) (FLOYD; JACOBSON, 1993) (GERLA et al., 2002) e necessidade de perda de pacotes para reconhecimento de congestionamento (HASEGAWA; MURATA, 2006). A estratégia vê a rede como uma caixa preta (JACOBSON, 1988) (JACOBSON, 1990), isto é, não retorna informações à origem. O TCP irá conduzir a uma perda de pacotes para assim tentar calcular a capacidade de rede (MASCOLO, 1999).

3. O ajuste não adequado entre a janela do TCP e a largura de banda da rede irá resultar na acumulação de grandes filas e perdas nos dispositivos de rede (KALAMPOUKAS et al., 2002).

4. O TCP não consegue distinguir se o pacote foi perdido por falha no enlace ou por descarte de pacotes devido a um congestionamento de rede (GERLA et al., 1999a) (JIANG et al., 2009); se a falha foi no enlace não há necessidade de ativar os mecanismos de controle de congestionamento.

5. O protocolo TCP não consegue oferecer um compartilhamento por igual de largura de banda quando os fluxos com diferentes RTTs competem em um mesmo ponto (MO; WALRAND, 2000) (GERLA et al., 1999a) (HASEGAWA; MURATA, 2006).

Várias pesquisas foram realizadas para tentar resolver os problemas enumerados acima.

Problema 1: Segundo Morris (MORRIS, 1997) a habilidade do TCP em prover um serviço eficiente em um ponto central de tráfego (*bottleneck*) diminui de acordo com o crescimento do número de fluxos. Seu trabalho afirma que para resolver tal problema deve-se fazer com que o TCP seja menos agressivo e mais adaptativo quando a janela de congestionamento for pequena.

Problema 2: A adoção de técnicas de gerenciamento ativo de filas (AQM) visam detectar uma situação de congestionamento antes que a fila do roteador fique cheia, além de prover o repasse da informação da possível ocorrência de congestionamento aos envolvidos na conexão. A notificação explicita de congestionamento (ECN), e a detecção aleatória antecipada (RED) foram criadas com o objetivo de serem usadas com os protocolos TCP/IP (FLOYD; JACOBSON, 1993) (RAMAKRISHNAN; FLOYD, 1999) (RAMAKRISHNAN et al., 2001). Floyd descreveu os benefícios e problemas na adoção do ECN no TCP; o uso (com o mecanismo DECbit) permite o reconhecimento de uma situação de congestionamento sem chegar ao ponto de ter que descartar pacotes; o ECN apresentou dois problemas: (1) mensagens ECN podem não chegar ao destino, por serem descartadas pela rede, evitando o recebimento da notificação de congestionamento, e (2) problemas de compatibilidade de dispositivos sem suporte a ECN com dispositivos com suporte (FLOYD, 1994). O RED tenta evitar possíveis congestionamentos com base no tamanho médio de fila no roteador. A notificação do congestionamento pode ser feita com a marcação de segmentos ou com o seu descarte. Uma desvantagem do RED é a necessidade do ajuste fino de seus parâmetros (FLOYD; JACOBSON, 1993).

Problema 3: Quando a conexão TCP é estabelecida por caminhos que apresentem esquemas de velocidade controlada (*rate-controlled*) e sem velocidade controlada (*nonrate-controlled*), caso não ocorra perdas de congestionamento o TCP irá aumentar sua janela até o tamanho máximo. A diferença entre a janela do TCP e a banda de rede disponível irá resultar na acumulação de grandes filas e perdas nos dispositivos de borda do esquema com velocidade controlada (KALAMPOUKAS et al., 2002). Kalampoukas e Varna montaram um esquema baseado na modificação do campo janela do TCP receptor. Os resultados demonstraram eficiência no controle de perdas, justiça e vazão (*throughput*) superiores a mecanismos de descarte como o RED (KALAMPOUKAS et al., 2002).

Problema 4: O TCP não funciona bem em redes que apresentam erros frequentes no meio físico (exemplo, redes sem fio), pois ao detectar uma perda de pacotes, o protocolo acredita que isso foi devido a um congestionamento, e ativa mecanismos para resolver o problema. Com a ativação de mecanismos, como o *slow-start*, haverá uma redução desnecessária de envio de dados (BALAKRISHNAN et al., 1995) (LEUNG; YEUNG, 2004) (ZHANG; FENG, 2009)

(VANGALA; LABRADOR, 2003). Uma forma bastante pesquisada na literatura para tentar resolver este problema é a utilização do TCP *Snoop*. O *Snoop* teve sua origem no trabalho de Balakrishnan *et al.* (BALAKRISHNAN et al., 1995) e foi pesquisado e expandido em diversos trabalhos (GARCIA et al., 2002) (ZHANG et al., 2008) (VANGALA; LABRADOR, 2003) (SUN et al., 2004) (ZHANG; FENG, 2009) (ZHANG et al., 2008) (GARCIA et al., 2002) (LEUNG; YEUNG, 2004) (MOON; LEE, 2006). Simplificadamente, a ideia consiste na instalação de um serviço no dispositivo que faz a ligação das redes sem/com fio. Este serviço tem por objetivo armazenar temporariamente pacotes TCP, e gerenciar as mensagens TCP do receptor, com isto, atuando no reenvio de segmentos perdidos, e na ocultação de segmentos de resposta desnecessários.

Problema 5: Um fator importante que deve ser mantido em dispositivos que trabalham com diversas conexões TCP é a justiça (*fairness*) [4]. Conexões com RTTs elevados tendem a possuir uma velocidade de transmissão inferior a outras conexões que compartilham uma mesma rota (FLOYD, 1991). Em (MO; WALRAND, 2000) foi criado um protocolo para controlar o tamanho da janela do TCP, baseando-se no tempo total, porém é necessária a intervenção do usuário no seu ajuste. O esquema *bandwidth aware TCP* (BA-TCP) (GERLA et al., 1999a) foi criado para ser usado em enlaces de satélites, e possibilita o envio da informação de RTT na camada de rede (campo no protocolo IPv6). O resultados demonstraram que a implementação fez uma alocação justa de largura de banda, e a vazão (*throughput*) foi três vezes maior que outras implementações TCP. O TCP Symbiosis (HASEGAWA; MURATA, 2006) faz a alteração de tamanho da janela do TCP de acordo com a largura de banda do caminho de rede, e com isso busca resolver o problema de injustiça causado pela diferença de RTT. Por análises matemáticas o TCP Symbiosis confirmou que tem uma melhor escalabilidade que outras implementações TCP.

2.4 GENERALIZED WINDOW ADVERTISING (GWA)

Alguns trabalhos tentam resolver mais de um dos problemas acima (GERLA et al., 1999a) (GERLA et al., 2002) (HASEGAWA; MURATA, 2006) (GERLA et al., 2000) (GERLA et al., 1999b) (GERLA et al., 2001). Em (GERLA et al., 2002) foi proposto um esquema, chamado *generalized window advertising* (GWA), para auxiliar no controle de congestionamento. O método faz uso do esquema *end-host-network* onde o congestionamento é notificado da rede para os pontos finais. Essa notificação é feita pelo repasse da informação de banda disponível para o cabeçalho IPv6 no sentido transmissor-receptor. O receptor irá

[4] A justiça (*fairness*) deve ser interpretada o uso por igual dos recursos da rede pelos nós que a compõe.

extrair a informação do protocolo IP e inserir no campo janela do TCP caso o valor seja menor que o valor atual. O GWA demonstrou ser mais eficaz em manter a estabilidade e justiça da rede comparado ao TCP com RED, e o ECN. Propostas semelhantes a (GERLA et al., 2002) foram feitas em (GERLA et al., 2000) e (GERLA et al., 1999b), onde além da rede fornecer a informação de banda disponível é também fornecido o tempo médio de propagação até a origem. Os resultados indicaram que a rede atingiu um maior equilíbrio e divisão justa de tráfego.

2.5 GERENCIAMENTO ATIVO DE FILAS

O gerenciamento de filas tradicional usa uma marca (*threshold*) para cada fila, quando o número de *bytes*/pacotes ultrapassar a marca os pacotes são descartados (*Tail Drop*). Em algumas situações esta técnica apresenta problemas que geram injustiça, como por exemplo, o uso total do enlace por um ou poucos fluxos (problema denominado *Lock-Out*). Outro tipo de problema é chamado de *Full Queues*, onde rajadas de dados (*burst*) ocupam todo o espaço da fila (THIRUCHELVI; RAJA, 2008).

O *Lock-Out* pode ser solucionado com o uso de disciplinas *Random Drop on Full* ou *Drop Front on Full* quando a fila estiver cheia. A solução para o *Full Queues* é obtida através do descarte de pacotes antes do enchimento da fila, o que é denominado gerenciamento ativo de filas (AQM). Métodos AQM são necessários até para roteadores que utilizam algoritmos de escalonamento por fluxo, como o *class-based queueing* (CBQ), pois estes escalonadores sozinhos não possibilitam controlar o tamanho total de uma fila (BRADEN et al., 1998).

Os esquemas AQM permitem a roteadores controlar o tráfego de rede pelo descarte de pacotes antes do transbordo da fila. A próxima seção apresenta o RED, um mecanismo que resolve os dois problemas citados acima.

2.5.1 RANDOM EARLY DETECTION (RED)

O RED deve ser utilizado em uma rede onde o protocolo de transporte responde a indícios de congestionamento na rede. Ele busca simplificar o trabalho do controle de congestionamento da camada de transporte. Embora não se aplique exclusivamente a redes TCP/IP, algumas características são exclusivas; a marcação ou o descarte de um pacote deve ser o suficiente para indicar a existência de um congestionamento.

O mecanismo de controle de congestionamento presente no RED monitora a média do tamanho da fila realizando um descarte probabilístico. Além do descarte, o algoritmo pode

marcar um pacote para descarte ajustando um *bit* do cabeçalho do protocolo de transporte; quando a fila excede um nível máximo, os pacotes não são marcados, mas descartados diretamente (FLOYD; JACOBSON, 1993).

2.5.1.1 FORMULAÇÃO

O algoritmo RED mantém uma média móvel do nível de utilização da fila, avg. A cada novo pacote, esta média é recalculada, e se a fila não estiver vazia, avg é atualizada pela equação seguinte.

$$\overline{avg} = (1-w)\, avg + w \cdot q \tag{2}$$

onde w é um fator de peso e q corresponde ao tamanho da fila. Caso a fila esteja vazia, a média é reduzida proporcionalmente ao tempo (t) em que ela esteve vazia (Equação 3).

$$\overline{avg} = (1-w)^t\, avg \tag{3}$$

A variável avg é utilizada para decidir se um pacote vai ser descartado. Os pacotes são descartados probabilisticamente usando a função de probabilidade definida como:

$$f = \begin{cases} 0, & avg \leq minth \\ \frac{avg - minth}{maxth - minth}\, maxp, & minth < avg < maxth \\ 1, & avg \geq maxth \end{cases} \tag{4}$$

onde $minth$ e $maxth$ são parâmetros para definição do tamanho da fila, e $maxp$ corresponde a probabilidade máxima (FLOYD; JACOBSON, 1993).

3 CONTROLE DE CONGESTIONAMENTO BASEADO NA BANDA DISPONÍVEL

O controle de congestionamento padrão do protocolo TCP apresenta várias características e problemas. Entre eles pode-se citar: a acumulação de grandes filas e perda de pacotes nos roteadores da rede; a necessidade da informação sobre perda de pacotes para o controle de transmissão; pacotes perdidos por falha no enlace ativam desnecessariamente os mecanismos de controle de congestionamento; e a não oferta de um compartilhamento justo de banda.

A literatura tem abordado tópicos que norteiam a criação de mecanismos que visam minimizar tais problemas. Alguns destes tópicos são recapitulados a seguir:

- Com o retorno explícito da banda disponível da rede, o emissor TCP pode controlar o tamanho da janela sem a necessidade da perda de pacotes devido ao congestionamento. A rede deve retornar informações sobre seus recursos para os emissores (*sources*) TCP (GERLA et al., 2000).
- A forma mais efetiva de detecção de congestionamento pode ocorrer no *gateway* (FLOYD; JACOBSON, 1993).
- A informação da largura de banda disponível pode dar um valor mais apropriado para a variável de tamanho limite da janela (*ssthresh*), melhorando a fase de *slow-start* do TCP. Quanto mais informações sobre as condições de rede forem disponíveis para o protocolo de transporte, mais eficiente podem ser as transferências de dados (ALLMAN; PAXSON, 1999).
- Se o transmissor TCP reconhecer a largura de banda (ter a informação), rapidamente e adequadamente ele pode criar um melhor mecanismo de controle de congestionamento (JIANG et al., 2009) (HASEGAWA; MURATA, 2006).

Estes tópicos conduziram a criação de um esquema baseado no envio da quantidade disponível de *bytes* da fila dos roteadores para os pontos finais TCP. Este esquema foi batizado

de *new generalized window advertising* (*NGWA*). Neste capítulo são descritos detalhes da proposta, bem como, sua análise.

3.1 ESQUEMA PROPOSTO

A implementação TCP por padrão utiliza o campo janela do segmento TCP recebido como parâmetro da rajada de dados a ser transmitida. O *NGWA* busca trazer até o receptor a quantidade de banda disponível na rede. O número total de *bytes* disponíveis na fila do nó [1] é armazenado na variável W_{ngwa}, sendo inserida na camada de rede; a variável é armazenada no campo *options* do cabeçalho IPv4. Desta forma é mantida a compatibilidade com o protocolo IP. A atualização da variável é realizada pelos nós da rota TCP. Cada nó pelo qual os pacotes trafeguem irá realizar a atualização da variável; caso $W_{ngwa}(i)$ seja menor que $W_{ngwa}(i-1)$ então $W_{ngwa} = W_{ngwa}(i)$.

A Figura 3 ilustra de maneira simples o processo. Internamente, quando um pacote chega ao receptor ele é processado normalmente, a variável W_{ngwa} é extraída do cabeçalho IP e depositada na memória. Durante a criação de um segmento ACK a ser enviado ao transmissor, W_{ngwa} é extraída da memória, podendo sofrer algum tipo de processamento, sendo inserida no campo janela do segmento TCP. Com isso o receptor TCP saberá a quantidade máxima de *bytes* que a rede suporta naquele instante.

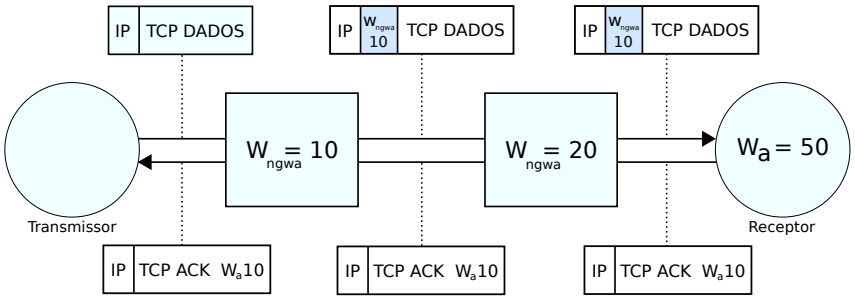

Figura 3: Esquema proposto.

Sabe-se que na maioria das aplicações que utilizam os protocolos TCP/IP um modelo cliente/servidor é utilizado — o receptor transmite poucos pacotes, e em sua maioria são de resposta (ACK). Usando o modelo cliente/servidor tem-se os papéis:

- Fonte: Faz o papel de servidor, transmitindo dados para o receptor. A taxa de transmissão será ajustada com o auxílio da variável W_a [2] do segmento TCP emitido pelo receptor.

[1] Este termo é genérico fazendo referência a um dispositivo com capacidades de roteamento.
[2] W_a indiretamente contém W_{ngwa} (Equação 6).

- Nós intermediários (roteadores): Realizam a criação/atualização da variável W_{ngwa}. Ela irá manter a menor quantidade de *bytes* disponível da fila de todos os nós intermediários.

- Receptor: Na recepção dos dados a variável W_{ngwa} é extraída do cabeçalho do pacote recebido. O valor de W_{ngwa} pode sofrer então algum tipo de processamento (ex. suavização). Após isso ela será utilizada como parâmetro para a criação do campo janela do segmento de resposta a ser transmitido.

Utilizando o mecanismo base acima, o método proposto neste trabalho foi dividido em oito modos (Tabela 1). Estes são constituídos de uma combinação das características: uso da janela de congestionamento, equação de suavização, e divisão por fluxos (DPF).

Tabela 1: Modos do *NGWA*

	Jan. de Congest.	Eq. Suavização	Div. por Fluxos
1	X	–	–
2	X	X	–
3	–	–	–
4	–	X	–
5	X	–	X
6	X	X	X
7	–	–	X
8	–	X	X

Na prática sabe-se que a implementação TCP é a mesma, não havendo diferença entre cliente e servidor. Por isto a variável W_{ngwa} é utilizada por ambos da mesma forma.

3.1.1 USO DA JANELA DE CONGESTIONAMENTO

Esta característica é aplicada apenas a fonte. Quando ativa, uma implementação TCP com controle de congestionamento é utilizada pelo transmissor, neste trabalho foi usado o TCP New Reno. Caso a característica não esteja ativa, naturalmente o TCP padrão (RFC 793) (POSTEL, 1981b) será utilizado. A Equação 5 define a taxa de transmissão em ambos os casos.

$$W_t = \begin{cases} min[W_a, W_c] - W_u, & \text{opção ativa.} \\ W_a - W_u, & \text{caso contrário.} \end{cases} \tag{5}$$

onde, W_u é a quantidade de dados enviados mas não confirmados.

É importante deixar claro que a variável W_{ngwa} é utilizada pelo lado receptor, sendo indiretamente aplicada ao transmissor pela utilização do parâmetro W_a, que é obtido do receptor. Este parâmetro é gerado pelo receptor com o uso da Equação 6.

$$W_a = \begin{cases} min[W_b, W_{max}, W_{ngwa}], & \text{com } NGWA. \\ min[W_b, W_{max}], & \text{sem } NGWA. \end{cases} \quad (6)$$

onde W_{max} corresponde ao tamanho máximo da janela; e W_b indica a quantidade de espaço disponível na memória do receptor TCP.

3.1.2 EQUAÇÃO DE SUAVIZAÇÃO

A janela do TCP em geral varia bastante em uma pequena porção de tempo (MOON; LEE, 2006). A equação de suavização (7) é utilizada pelo receptor TCP final, visando evitar alterações bruscas na janela de transmissão, sendo $0 < \alpha < 1$.

$$\overline{W}_{ngwa} = (1-\alpha) \cdot \overline{W}_{ngwa} + \alpha \cdot W_{ngwa} \quad ^{(nota)\ 3} \quad (7)$$

3.1.3 DIVISÃO POR FLUXO (DPF)

Por padrão o *NGWA* armazena na variável W_{ngwa} a quantidade total de *bytes* disponíveis na fila dos nós intermediários. Este comportamento pode trazer injustiças, pois um fluxo pode consumir todos os recursos do nó.

A divisão por fluxos (DPF) realiza o compartilhamento dos *bytes* livres na fila do nó pela quantidade de fluxos/conexões ativas. A detecção de fluxos TCP é feita através do registro de origem/destino de endereços IP e portas.

Quando esta opção está ativa, um processo é executado para criar e manter o registro de conexões ativas que trafeguem pelo nó (variável nf). O valor de W_{ngwa} é calculado então pela equação (8), onde B_r é o espaço total disponível na fila do roteador. O fator $0,98$ é utilizado para prever espaço na fila para os quadros de confirmação (ACKs).

$$W_{ngwa} = \frac{0,98 \cdot B_r}{nf} \quad (8)$$

A característica de divisão por fluxos é ideal quando o roteamento é estático, se ele for dinâmico (onde pacotes podem trafegar por caminhos diversos) a quantidade de *bytes* (W_{ngwa}) não será corretamente estimada. Isso pode gerar certa injustiça, porém não irá alterar de modo

[3] A aplicação da equação pode ter um melhor efeito se a sua aplicação for condicionada. Por exemplo, nos intervalos de tempo em que a rede está fortemente utilizada e, portanto, a quantidade livre de *bytes* na rota é próxima de zero, poder-se-ia evitar a suavização (utilizando diretamente o valor W_{ngwa}) e aplicá-la somente quando W_{ngwa} aumenta bruscamente (evitando um aumento brusco na taxa de transmissão do TCP).

significativo o desempenho do sistema.

3.2 PROPOSTA SEMELHANTE

Na literatura existe um esquema semelhante ao proposto, o *generalized window advertising - GWA* (GERLA et al., 2002). Porém, o *NGWA* foi previamente desenvolvido sem o conhecimento da existência do *GWA*, o que o torna diferente em vários pontos:

1. O *GWA* é ideal para filas baseadas em fluxo (*per-flow queuing*) com o algoritmo *round robin* (GERLA et al., 2002). O *NGWA* é independente de tipos de filas pois atua após a saída do pacote da fila.

2. No *GWA* perdas não são previstas, mas podem ocorrer. Na sua ocorrência são ativados mecanismos de controle de congestionamento, como o *slow-start* (GERLA et al., 2002). O *NGWA* tem modos que não utilizam tais mecanismos.

3. O cálculo de número de fluxos ativos do *GWA* é uma estimativa obtida pelo sistema proposto (GERLA et al., 2002). O número de fluxos do *NGWA* é calculado pelo próprio método com base no registro das conexões que fazem uso do roteador.

3.3 IMPLEMENTAÇÃO

Para realizar a validação da proposta, optou-se por utilizar um simulador de rede, devido ao seu amplo uso em trabalhos relacionados (MAN et al., 2006) (KODAMA et al., 2009) (GERLA et al., 2000) (JAIN; DOVROLIS, 2003) (GERLA et al., 2002) (HASEGAWA; MURATA, 2006) (LOW et al., 2001). O simulador referido e utilizado neste trabalho é o NS (NS-3, 2011). Uma escolha natural, devido a sua popularidade, seria a utilização do *NS-2*. Porém esta versão não fornece suporte ao desenvolvimento da proposta, pelos seguintes motivos:

- Falta da implementação da leitura do campo janela do cabeçalho TCP recebido (NS-2-LIMITATIONS, 2011).

- Ausência de realismo básico nas simulações. O conceito de IPv4 e IPv6 não existe — não se faz uso de endereços IP.

- O *NS-2* foi desenvolvido no período 1997-2000 e está desatualizado e não mantido (NS-2-OUTDATED, 2011).

- Necessidade de desenvolvimento das simulações utilizando *scripts Tcl*. Isso acarreta dificuldades no processo de depuração e perda de performance.
- Para se extrair dados das simulações necessita-se escrever funções específicas.

Optou-se pelo uso do *NS-3* (NS-3, 2011). Este é um eventual substituto para o *NS-2*, tendo seu surgimento no ano de 2006. Ele é um *software* livre (GLPv2), código totalmente feito em linguagem *C++*, simulador discreto, mas com foco para o realismo, apresentando flexibilidade de extração de dados. No *NS-3* as simulações podem ser escritas em linguagem *C++* ou *Python*.

Um trabalho recente (WEINGARTNER et al., 2009) avaliou a performance de diferentes simuladores de rede, entre eles o NS-{2,3}. O *NS*-3 foi considerado superior em performance sobre os demais simuladores.

Tinha-se em mente realizar as simulações da proposta utilizando o protocolo IPv6. Porém, o *NS-3* ainda não oferece suporte a IPv6 + TCP (NS-3-IPv6, 2011). Logo, o protocolo IPv4 foi utilizado.

A variável W_{ngwa} é armazenada no campo *options* do cabeçalho IPv4. Desta forma é mantida a compatibilidade com o protocolo IP. A atualização da variável é realizada pelos nós do caminho TCP. Implementou-se este mecanismo após a retirada do pacote da fila do nó (Figura 4). O algoritmo está presente na classe *NgwaQueue* e é baseado na classe *DropTailQueue* do *NS-3*. Foi mantida a sua estrutura básica, adicionando o algoritmo para realizar a atualização do campo W_{ngwa}.

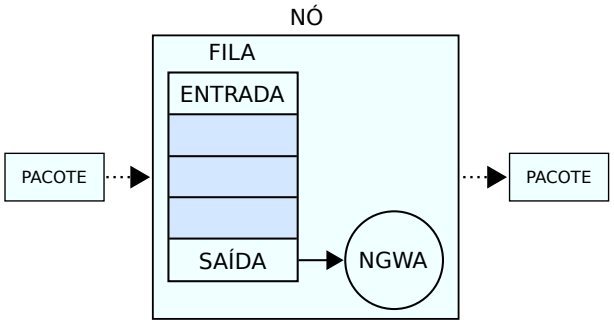

Figura 4: Caminho do pacote em um nó. Após o pacote sair da fila é que ele será atualizado pelo *NGWA*.

Internamente, quando um pacote chega ao receptor ele é processado normalmente, a variável W_{ngwa} é extraída do cabeçalho IP e depositada na memória. Na criação de um segmento a ser enviado ao transmissor a variável W_{ngwa} é utilizada pelo *NGWA* de acordo com

a Equação 6. Após este processo, o resultado é inserido no campo janela do segmento TCP a ser transmitido.

Ao executar uma simulação, o modo do *NGWA* é selecionado através da alteração de uma variável de controle. O código da implementação foi projetado para tratar os modos de maneira modular. Também com objetivos de avaliação, o *NGWA* pode ser desativado com o ajuste de uma variável, com isso o TCP/IP volta a operar da maneira tradicional.

3.4 RESULTADOS

Como dito, as simulações foram executadas no simulador de rede *NS-3*. Foram criadas duas topologias para testes, uma simples (poucos nós) e outra com um maior número de nós e enlaces. A primeira topologia (I) foi baseada no trabalho de (FALL; FLOYD, 1996); Figura 5. Nela têm-se dois enlaces, o nó $N1$ tem ligação com o nó $N2$, e este com nó $N3$. Os enlaces possuem 10 $Mbps$ com 10 ms de latência, e 1 $Mbps$ com 50 ms, respectivamente. Foi verificado em trabalhos relacionados ((LOW et al., 2001) (GERLA et al., 2000) (GERLA et al., 1999a) (FLOYD, 1994) (GERLA et al., 1999b) (MO; WALRAND, 2000) (YANG; LAM, 2000) (MASCOLO, 1999) (HASEGAWA; MURATA, 2006) (MORRIS, 1997) (LOW et al., 2001)) a utilização de uma topologia em comum: Figura 6. Nesta segunda topologia (II), fluxos TCP são estabelecidos entre os nós posicionados nas extremidades da rede.

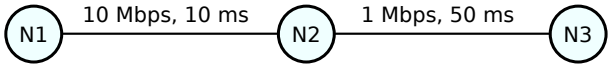

Figura 5: Topologia I.

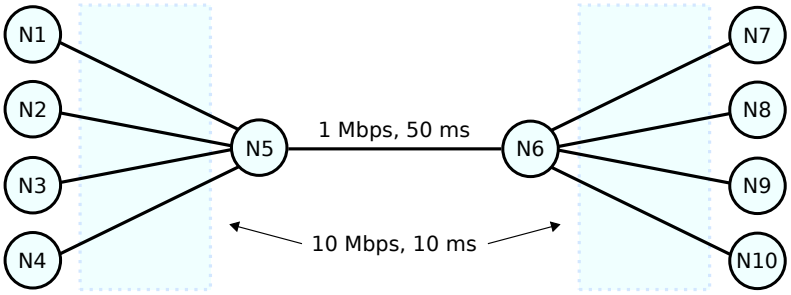

Figura 6: Topologia II.

As simulações executadas tiveram por objetivo avaliar o desempenho da proposta. Foram executados testes de taxa de transmissão de fluxos e justiça (*fairness*) de uso de rede, variação da janela de transmissão, taxa de pacotes perdidos, e utilização de fila.

Para realizar uma comparação com a proposta foram utilizadas as implementações TCP New Reno e a TCP padrão (POSTEL, 1981b). Todos os modos do *NGWA* foram avaliados.

Os seguintes valores foram considerados: W_{max} = 64 *kbps*; tamanho dos segmentos TCP: 1000 *bytes*; fila: *drop tail*; capacidade da fila dos roteadores: 97 *kBytes*; constante $\alpha = 0.3$; tempo de simulação: dez minutos (após isso houve um comportamento estável). O tráfego da simulação foi gerado com a utilização do modelo *OnOffApplication* presente no *NS-3*. Este modelo foi configurado para transmitir constantemente dados até o tempo final da simulação, a uma taxa de 500 *kbps*.

3.4.1 VARIAÇÃO DA JANELA DE TRANSMISSÃO

Foi avaliada a variação da janela de transmissão para as duas topologias. Foram executados testes com o TCP padrão, TCP New Reno e com o *NGWA*.

Os resultados obtidos das topologias são bem diferentes. A primeira, devido a sua configuração, não deve levar a uma situação de congestionamento, com isso foi testado o comportamento do *NGWA* em uma rede estável. Na Figura 7 pode-se observar a taxa constante da janela de transmissão para as implementações testadas, o que representa uma ausência de perdas.

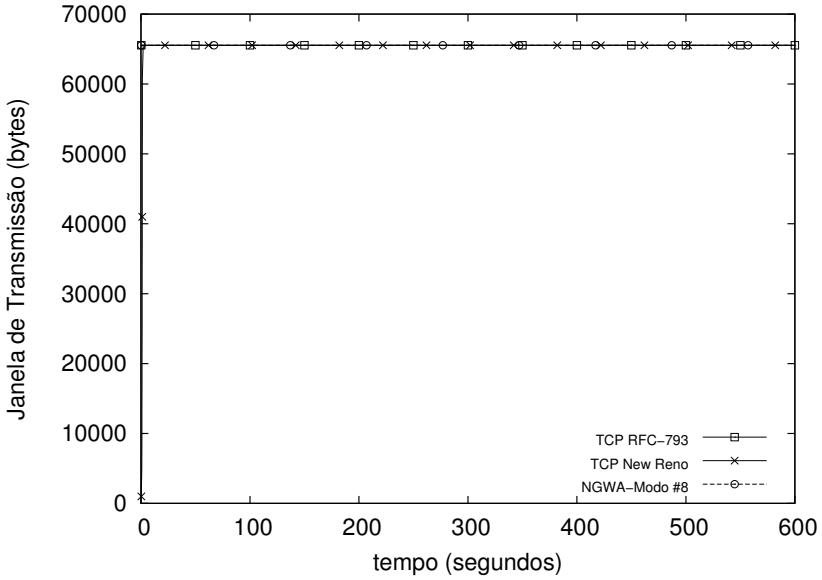

Figura 7: Variação da janela de transmissão da topologia I.

A segunda topologia possui mais nós e fluxos, tendo os dados trafegando por um *backbone* limitado a 1 *Mbps*. Nesta simulação cada nó do lado esquerdo estabeleceu uma conexão TCP com um nó do lado direito.

Houve variação do número de conexões ativas no tempo. Ao início uma conexão foi estabelecida, e a cada intervalo de 15 segundos uma nova conexão foi iniciada; no intervalo de

tempo de 45 até 120 segundos haviam quatro fluxos ativos; tendo dois fluxos ativos no tempo de 121 segundos, permanecendo assim até o final da simulação. Utilizando esta sequência foi possível de verificar a variação da janela de transmissão ao início/encerramento das conexões (Figura 8).

Os resultados da Figura 8 são analisados em três partes: (1) de 0s-45s, (2) de 60s-120s, e (3) de 121s-180s. Na primeira parte pode-se observar que o TCP New Reno inicia a janela de transmissão com um valor baixo, crescendo constantemente até atingir um pico, permanecendo nele até aos 30 segundos (aproximadamente), reduzindo a taxa, permanecendo em oscilação constante. O *NGWA* modos 5 e 6 seguiram o comportamento inicial do Reno, já os modos 7 e 8 iniciaram com um valor alto, após isso ambos os modos foram semelhantes: por volta dos 15 segundos a janela diminuiu chegando a 50000 *bytes* permanecendo constante até o estabelecimento de uma nova conexão, e assim a cada nova conexão existe uma queda na taxa de transmissão seguida por uma estabilidade. Na parte (2) o New Reno manteve seu comportamento oscilatório, enquanto o *NGWA* persistiu com a janela estável. Na última parte, tendo a queda de quatro para dois fluxos ativos, o TCP New Reno teve um crescimento na variação da janela, enquanto o *NGWA*, rapidamente, reduziu o tamanho da janela, mantendo-a estável até o final da simulação.

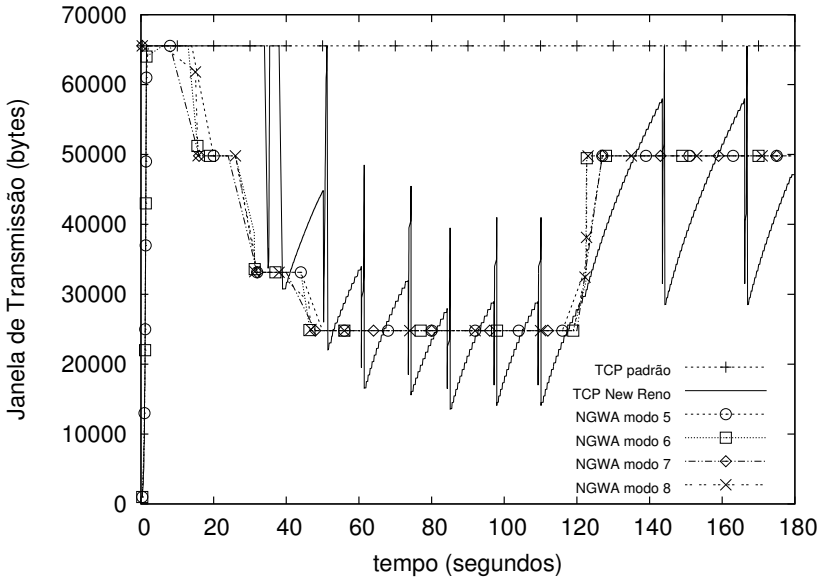

Figura 8: Variação da janela de transmissão da topologia II. Para observar a variação até o final da simulação, o gráfico apresenta o fluxo que permaneceu ativo do início ao final dela.

Nota-se que, em geral, o *NGWA* obteve uma taxa de transmissão estável, com grande justiça (*fairness*) na divisão de recursos entre os fluxos. O New Reno apresentou injustiça (*unfairness*) entre os fluxos e alta oscilação.

3.4.2 TAXA DE TRANSMISSÃO POR FLUXOS / JUSTIÇA DE USO DA REDE

Neste teste foi comparada a taxa de transmissão em *bytes/seg.* por cada fluxo. Com a análise deste resultado é possível verificar a justiça do uso de rede.

A topologia I possuí apenas um fluxo, por isso não foi utilizada neste teste. As Figuras 9 e 10 exibem a taxa de transmissão dos quatro fluxos da topologia II. Nesta topologia cada nó do lado esquerdo estabeleceu uma conexão TCP com um nó do lado direito, onde as fontes *on/off* estão transmitindo constantemente a uma taxa de 500 *kbps*, durante 10 minutos. Os dados obtidos foram analisados com base no método "*batch means*"(CHIEN, 1994), com 30 "*batches*", para a obtenção do intervalo de confiança de 95% presentes nos gráficos. Foram testados o TCP padrão (RFC 793), TCP New Reno, e todos os modos do *NGWA*.

Analisando os resultados percebe-se que o TCP padrão, naturalmente, foi o mais injusto - os fluxos 2 e 4 consumiram toda a banda de rede, deixando os fluxos 1 e 3 com uma baixa transmissão. O TCP New Reno manteve os fluxos com uma boa justiça. Os modos 1 e 3 do *NGWA* foram injustos, ao contrário, os modos 5, 6, 7 e 8 foram justos, chegando a superar o TCP New Reno. A equação de suavização teve efeito mais interessante nos modos 2 e 4, pois reduziu a injustiça presente nos modos 1 e 3.

3.4.3 TAXA DE PACOTES PERDIDOS

A quantidade de pacotes enviados e não entregues é um importante fator para verificação da qualidade do método de controle de congestionamento, pois, pacotes perdidos utilizam a rede, podendo aumentar o congestionamento. Assim como um *timeout*, o recebimento de três ACKs duplos (*triple dupacks*) é considerado um sinal de congestionamento na rede (STEVENS, 1997).

Durante as transmissões foram coletadas a quantidade total de *triple dupacks* e o número de perdas de todos os nós. Os resultados aqui apresentados são da topologia II. O acumulo de *triple dupacks* ao longo da transmissão é apresentado na Figura 11.

O número de *triple dupacks* do TCP padrão e do New Reno foram os mais altos da simulação. Os modos do *NGWA* tiveram diferentes desempenhos. O pior caso foi com o modo 3, atingindo 5279 *triple dupacks* recebidos. Seguido pelo modo 4 (2833), e 1 (30). A equação de suavização novamente apresentou um bom efeito para os modos 2 e 4, tendo um resultado muito superior aos modos 1 e 3. Os modos 2/5/6/7/8 não aparecem no gráfico, pois não receberam *triple dupacks*.

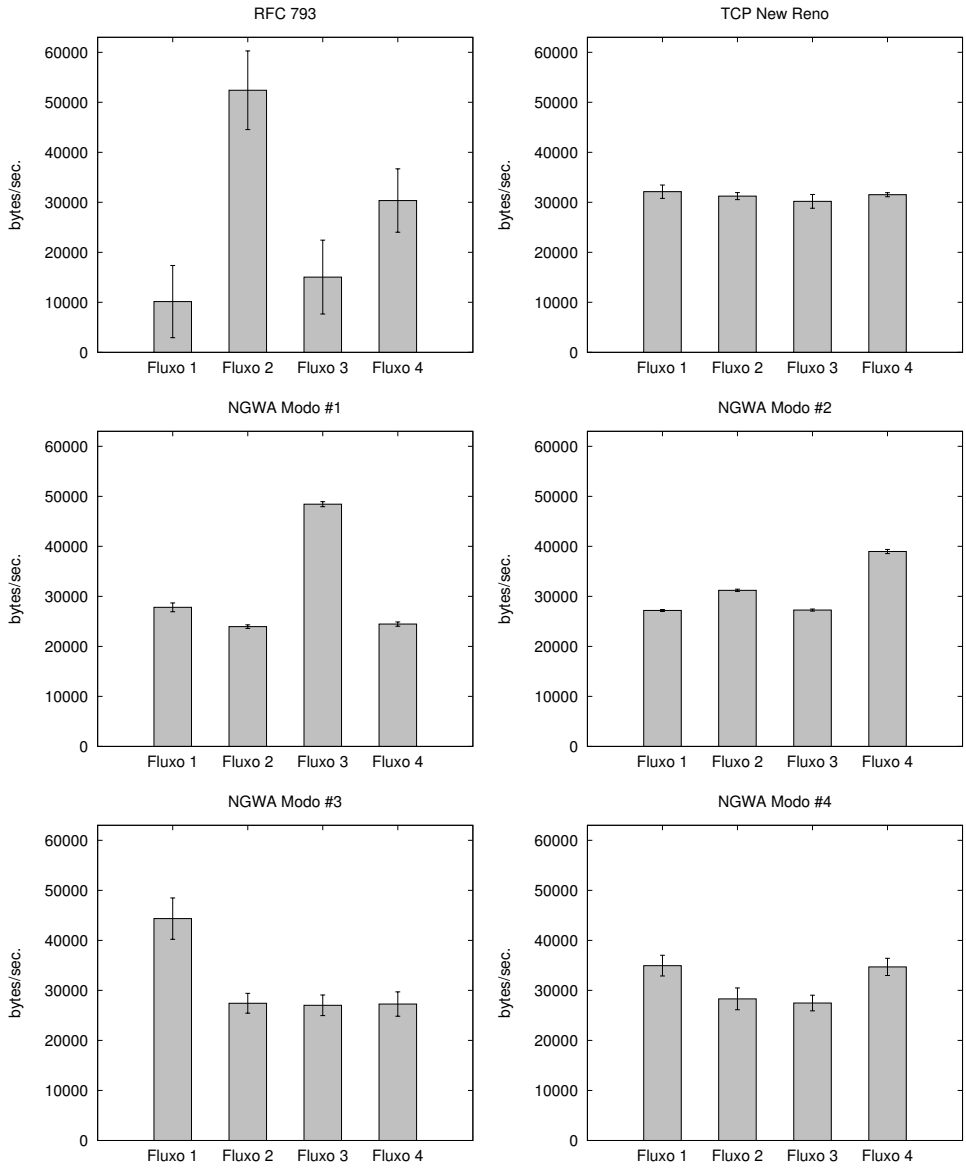

Figura 9: Taxa de transmissão dos quatro fluxos da topologia II para o TCP padrão (RFC 793), TCP New Reno, e *NGWA* modo 1 até o 4.

Também foi coletada a quantidade de pacotes perdidos, sendo apresentada na Tabela (2) a média para 30 rodadas. O TCP padrão produziu a maior quantidade média de perdas (4001), seguido por: *NGWA* modo 3 (1063), modo 4 (869), New Reno (328) e *NGWA* modo 1 (7). Os modos 2/5/6/7/8 não produziram perdas.

3.4.4 UTILIZAÇÃO DE FILA

Neste teste foi coletada a quantidade de *bytes* na fila do *backbone* a cada segundo até o término da simulação. As transmissões ocorreram nos quatro nós da topologia II, tendo-se quatro fluxos ativos até o final da simulação. Nas implementações TCP padrão e New Reno

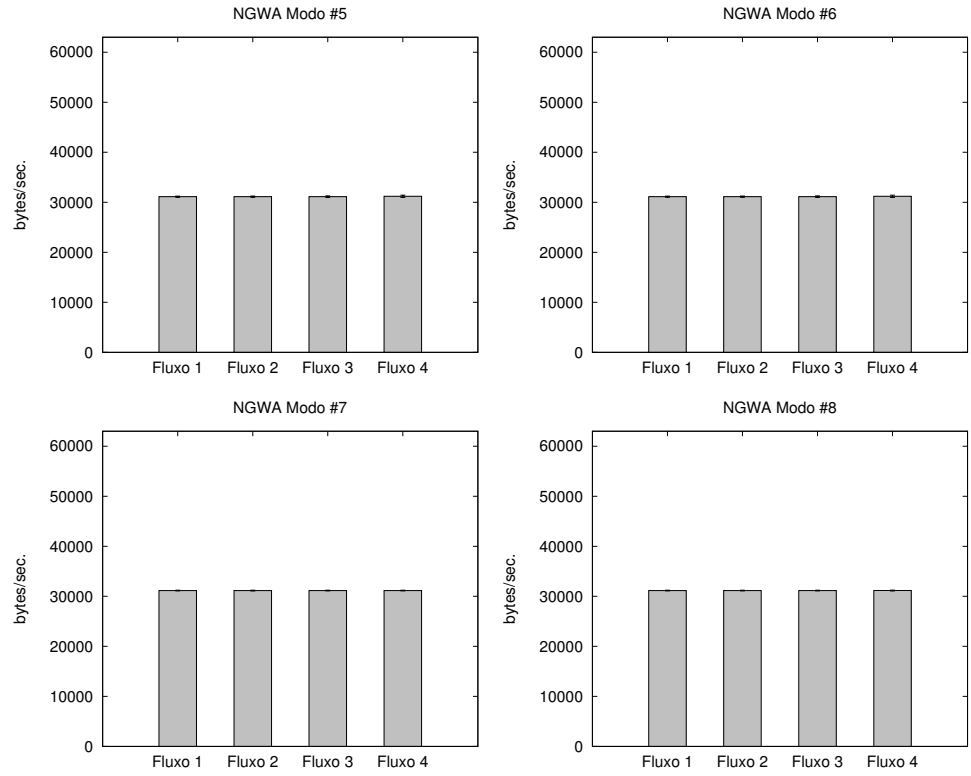

Figura 10: Taxa de transmissão dos quatro fluxos da topologia II para o *NGWA* modo 5 até o 8.

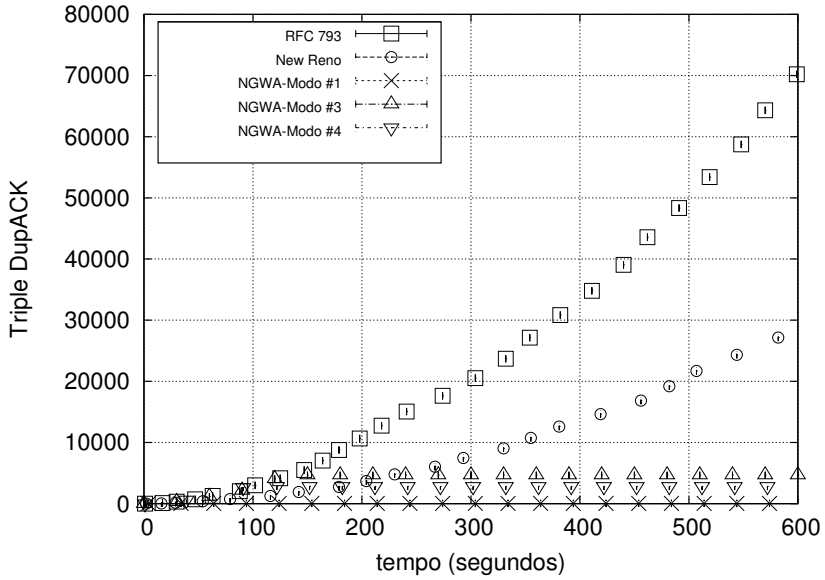

Figura 11: Acumulo de *triple dupack* durante a transmissão.

foram utilizados dois métodos de gerenciamento de filas, o *Drop Tail (FIFO)* e o RED. Os parâmetros para o RED foram: *minth* = 3 kBytes, *maxth* = 48,5 kBytes e *maxp* = 0,02 (WILLE et al., 2004).

As Figuras 12, 13, e 14 apresentam a utilização da fila (*bytes*) em 600 segundos de

Tabela 2: Média do número de pacotes perdidos em 30 rodadas.

	Topologia I	Topologia II
RFC 793	0	4001
New Reno	0	328
NGWA Modo #1	0	7
NGWA Modo #2	0	0
NGWA Modo #3	0	1063
NGWA Modo #4	0	869
NGWA Modo #5	0	0
NGWA Modo #6	0	0
NGWA Modo #7	0	0
NGWA Modo #8	0	0

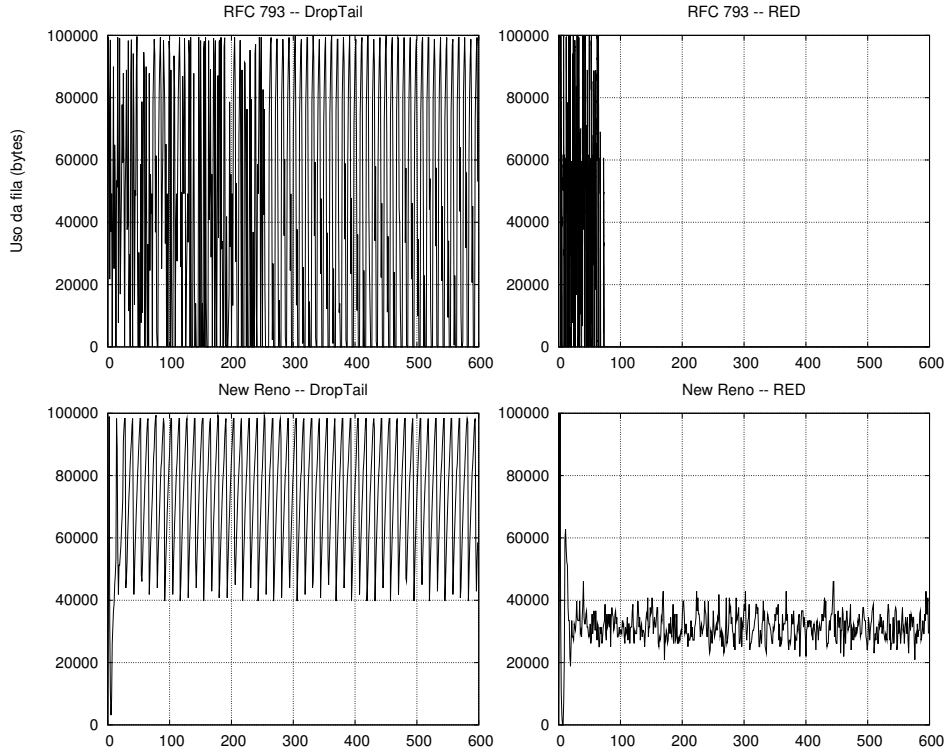

Figura 12: Intervalo de 600 segundos do nível de utilização da fila do *backbone* para o TCP padrão, e TCP New Reno.

simulação. Analisando o tempo integral (600 segundos) o *NGWA* modos 5-8 manteve uma maior utilização da fila; comparando os modos 1/3 com o 2/4 nota-se que a suavização fez o seu papel se adaptando gradualmente as alterações de rede. O TCP padrão com a fila *Drop Tail* manteve uma constante oscilação durante todo o período de simulação, usando o RED o resultado foi completamente diferente: a fila manteve alto uso até os 65 (aproximado) segundos iniciais, após isso a fila ficou vazia até o final da simulação; três dos quatro fluxos ficaram com a memória (*buffer*) de transmissão TCP constantemente cheia, e o último fluxo utilizou os recursos da rede. O TCP New Reno com o *Drop Tail* seguiu seu comportamento

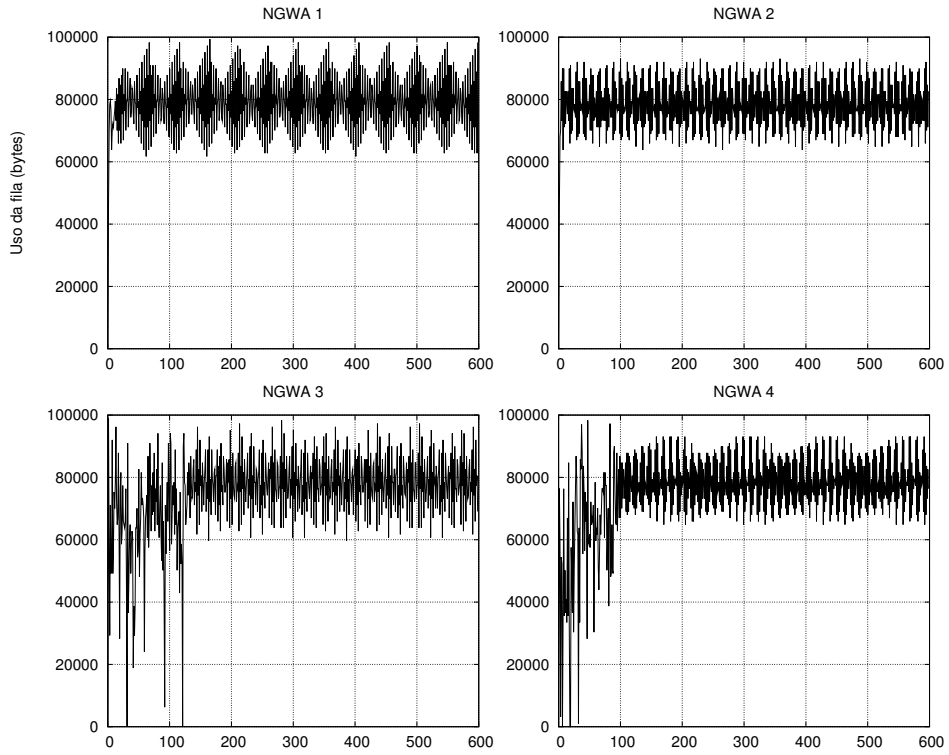

Figura 13: Intervalo de 600 segundos do nível de utilização da fila do *backbone* para os modos 1 à 4 do *NGWA*.

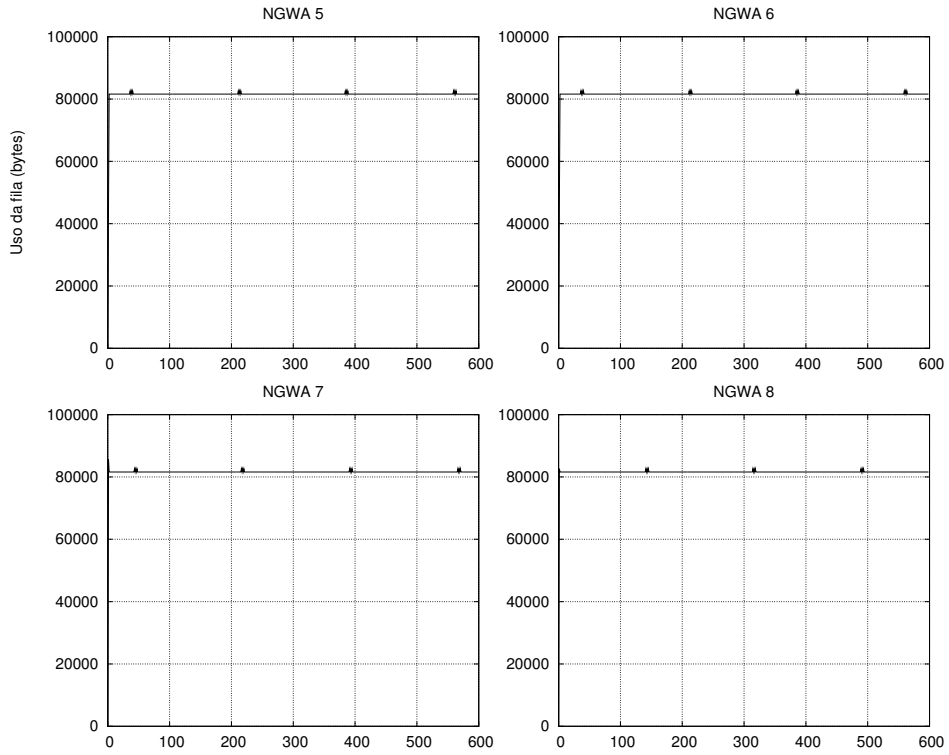

Figura 14: Intervalo de 600 segundos do nível de utilização da fila do *backbone* para os modos 5 à 8 do *NGWA*.

oscilatório típico, com a aplicação do método AQM RED a média de utilização da fila baixou para aproximadamente metade do obtido com o esquema anterior, reduzindo consideravelmente a variação da taxa de transmissão.

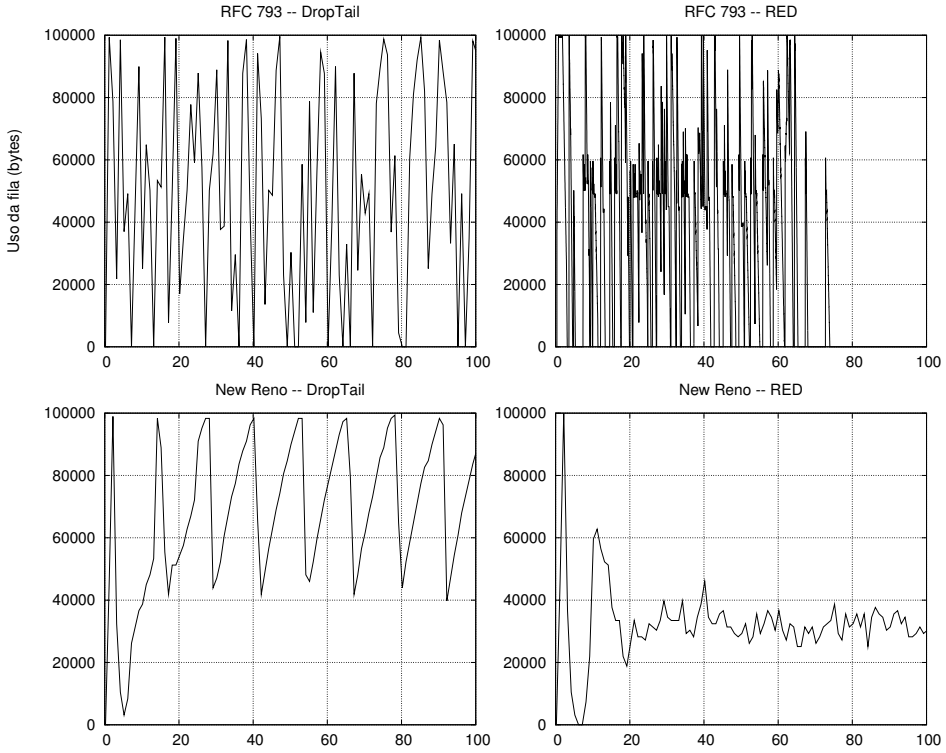

Figura 15: Intervalo de 100 segundos do nível de utilização da fila do *backbone* para o TCP padrão, e TCP New Reno.

As Figuras 15, 16, e 17 apresentam o nível de utilização da fila em até os 100 segundos de simulação. Para analisar o comportamento inicial, foi feito o registro dos 100 primeiros segundos de simulação. Neste, os modos 5-8 logo ao início da simulação atingiram uma estabilidade; os modos 1-4 sofreram oscilações, tendo os modos 2/4 com uma quantidade menor.

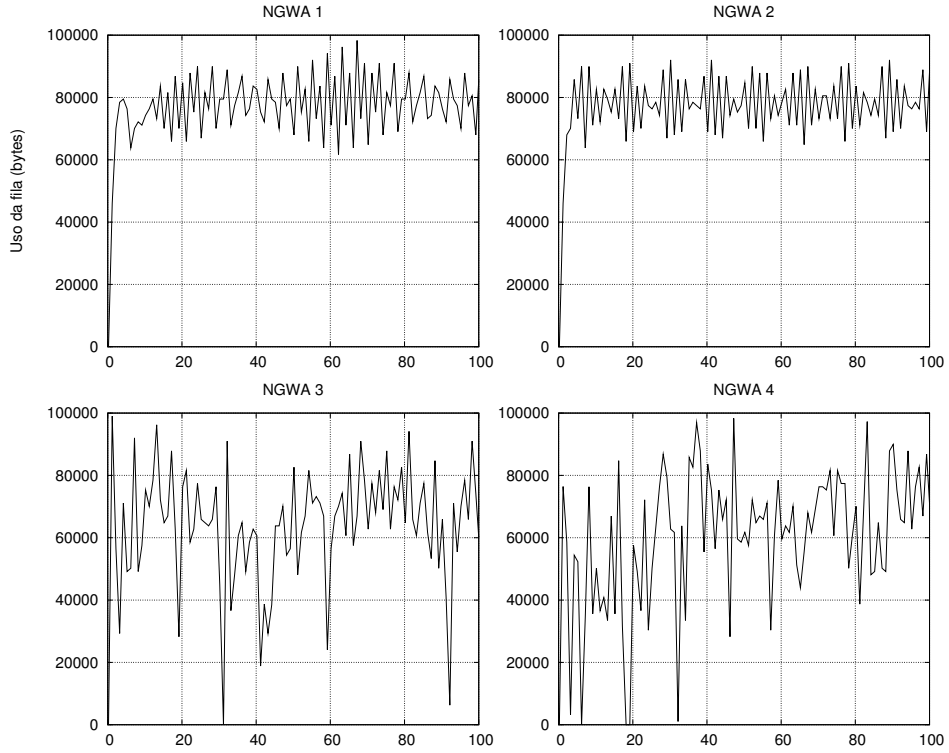

Figura 16: Intervalo de 100 segundos do nível de utilização da fila do *backbone* para os modos 1 à 4 do *NGWA*.

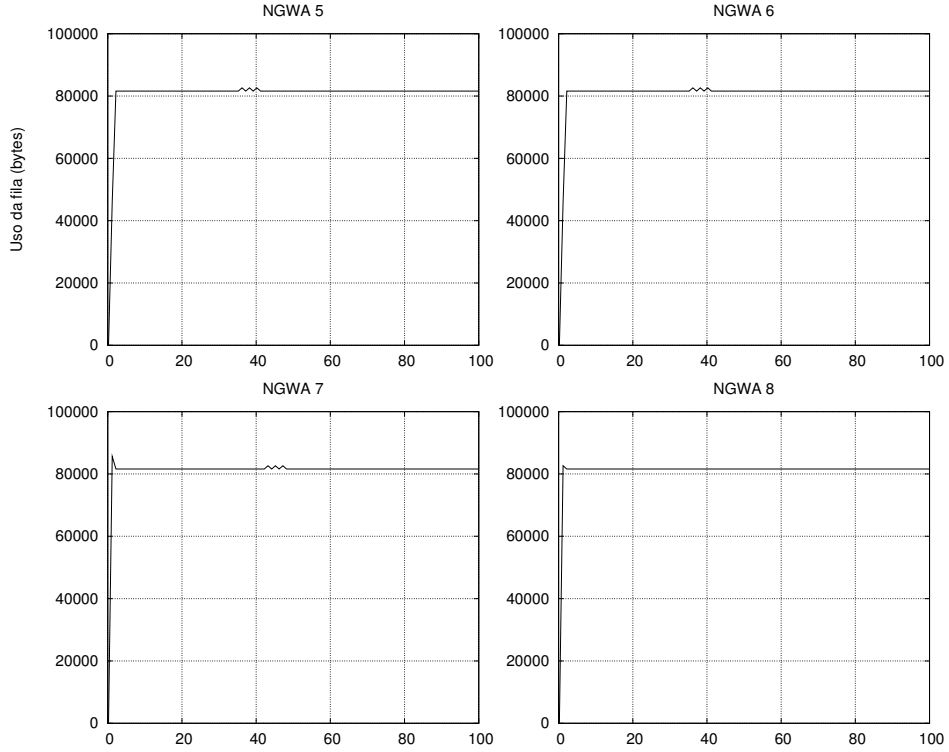

Figura 17: Intervalo de 100 segundos do nível de utilização da fila do *backbone* para os modos 5 à 8 do *NGWA*.

4 IMPLEMENTAÇÃO DO *NGWA* NO LINUX

O desempenho do *NGWA* no simulador *NS-3* foi satisfatório, o que conduziu a uma implementação em um ambiente real. Pelo grande uso no meio científico, ampla documentação e código escrito em linguagem C, optou-se pela escolha do Linux.

O Linux foi criado no ano de 1991 por Linus Torvalds como um sistema aberto e livre, por isso *hackers* utilizando a Internet puderam (podem) ajudar a desenvolver este *kernel* [1]. Sendo o Linux um *software* livre tendo a GPLv2 (*general public license*) como licença, o seu uso pelo meio acadêmico é favorecido, pois o código-fonte completo do sistema está disponível com permissão para alteração e compartilhamento de forma aberta (LOVE, 2010).

Além do Linux ser um *software* livre, é importante destacar algumas de suas características:

- Atualmente ele pode ser executado na arquitetura Alpha, ARM, PowerPC, SPARC, x86-64, além de muitas outras arquiteturas (LOVE, 2010).
- Diferentemente de outros projetos, como a linguagem Lua (Lua, 2012), o Linux tem seu desenvolvimento aberto. Caso alguém desenvolva um código que considere útil e ache adequado seu uso no Linux, pode enviá-lo à lista de desenvolvimento.
- Na versão 2.6.13 foi introduzido o conceito modular para o controle de congestionamento; que é uma interface definida para algoritmos de controle de congestionamento. Na versão 2.6.16-3 o Linux incorpora nove algoritmos de controle de congestionamento, como por exemplo, o TCP Vegas, TCP Westwood e TCP BIC (WEI; CAO, 2006).

O *NGWA* foi implementado no Linux versão 2.6.34 (Linux, 2011), esta versão foi escolhida por ser recente, pela sua maturidade e material bibliográfico disponível. Foi seguido o modelo de desenvolvimento do Linux no porte do *NGWA*; os códigos foram posicionados no subsistema de rede na forma de módulo e *patches* (porções de código).

[1] *Kernel* é um *software* que interage diretamente com o *hardware*, e estas interações geralmente são originadas por usuários do sistema operacional através de chamadas de sistema. Linux é um *kernel*, por isso neste trabalho *kernel* é tratado como seu sinônimo.

Na implementação do *NGWA* no Linux os códigos foram naturalmente divididos em dois grupos: fila e TCP. O primeiro foi construído em forma de módulo, e deve ser utilizado nos roteadores da rede, já o segundo foi embutido na implementação TCP do *kernel*. Atualmente não é possível de se utilizar o TCP padrão no Linux, pois os códigos de controle de congestionamento estão embutidos no núcleo, por isso o *NGWA* não pode ser utilizado com o TCP padrão. O código fonte completo da implementação se encontra no Anexo A.

Este capítulo está organizado em Seções. Nas Seções 4.1 e 4.2 são detalhados as implementações dos módulos Fila e TCP. A Seção 4.3 apresenta os resultados e as análises.

4.1 FILA NGWA

Ela atua nos roteadores da rede, não sendo necessária sua execução nos pontos finais TCP. Os pacotes recebidos são processados pelo roteador, sendo encaminhados para a fila *NGWA*. Resumidamente a fila realiza as seguintes operações:

- Verifica se o pacote recebido é do tipo TCP.
- Checa a existência do *NGWA* no campo *options* do cabeçalho IP, (1).
- Obtém a quantidade média de *bytes* disponíveis na fila, (2).
- Registra no campo *options* o menor valor entre (1) e (2).

Sua implementação no *kernel* foi feita na forma de módulo. Embora o Linux seja executado em apenas um espaço de endereçamento, ele é modular em dois aspectos: programação e execução. O primeiro diz respeito à organização dos códigos-fontes em arquivos que visam realizar uma tarefa de um domínio específico. O aspecto de execução refere-se a inserção/remoção dinâmica em tempo de execução do módulo no *kernel*, neste aspecto o código-fonte (sub-rotinas, dados...) é agrupado em um arquivo binário chamado *kernel object* (objeto do *kernel*). O suporte a módulos permite que o *kernel* seja mínimo, com recursos opcionais e *drivers* disponíveis em objetos separados. Os módulos também trazem a vantagem de permitir a atualização de códigos em tempo de execução, facilitando o processo de debugação e a carga de novos *drivers* sob demanda (LOVE, 2010).

A fila *NGWA* foi escrita dentro do subsistema de rede no arquivo *net/sched/sch_ngwa.c*. Os módulos de gerenciamento de filas presentes no diretório *net/sched* seguem um padrão de interface para funções e estruturas. As principais funções são: *init*, *enqueue*, *dequeue* e *drop*. A primeira é chamada na inicialização da fila, normalmente após o carregamento do módulo.

Nela deve-se ajustar o tamanho da fila (em pacotes), alocar memória para a fila, e inicializar estruturas específicas. As funções *enqueue* e *dequeue* fazem o trabalho de armazenar e remover um pacote da fila; neste processo podem ser feitas operações estatísticas, como a atualização da quantidade de pacotes e *bytes* na fila. Quando a fila está cheia, ou o algoritmo necessita descartar um pacote, a função *drop* é chamada. As estruturas *Qdisc_class_ops* e a *Qdisc_ops* fazem a ligação de funções e estruturas específicas do módulo com o *kernel*.

As rotinas específicas do *NGWA* são chamadas ao final da função *dequeue*. A Figura 18 exibe o pseudocódigo da fila *NGWA*. Na linha 1 é checado o tipo do pacote, que é obtido através da leitura do campo protocolo do cabeçalho IP, caso o pacote não seja do tipo TCP, nada será feito. Na sequência é checado o estado da opção DPF (descrita na Seção 3.1.3), estando ativada o código das linhas 4 à 12 é executado. Após isso é obtida a quantidade de *bytes* disponíveis na fila, e é checada a existência do *NGWA* no campo *obtions* do cabeçalho IP, existindo seu valor é armazenado. Quando o pacote não tiver registro *NGWA* ou quando a quantidade de *bytes* da fila do roteador (linha 13) for menor que o *NGWA* embutido no pacote, será inserida/atualizada a variável W_{ngwa} (Seção 3.1) do campo *options* do cabeçalho IP.

```
1   se (tipo_pacote != TCP)
2       não_processar
3   se (dpf) {
4       ajuste_ponteiro_transporte ()
5       tcph = cabeçalho_tcp
6       iph = cabeçalho_ip
7       se (tcph.flags.syn) {
8           se (nova_conexão (iph, tcph))
9               registrar_conexão (iph, tcph)
10      } se (tcph.flags.fin OU tcph.flags.rst)
11          remover_conexão (iph, tcph)
12  }
13  bytes_fila = obter_bytes_fila ()
14  bytes_ngwa = obter_ngwa_pacote (pacote)
15  se (bytes_fila < bytes_ngwa) OU (bytes_ngwa = 0)
16      adicionar_ngwa_opt (bytes_fila)
```

Figura 18: Pseudocódigo da fila *NGWA* no Linux.

Nas Seções que seguem são detalhados os principais códigos da Figura 18.

4.1.1 ENCAMINHAMENTO

O encaminhamento é realizado quando um pacote passa por um roteador, ou seja, ele recebe o pacote e encaminha para o próximo ponto; quando um pacote chega ao destino final

ele é recebido e tratado por camadas de rede superiores. Neste cenário é natural que um pacote encaminhado seja tratado de forma diferente a um recebido. Para realizar o encaminhamento não se necessita muito mais do que um pouco de memória, checagem do cabeçalho IP e retransmissão do pacote. No recebimento o processo é diferente, é necessário receber o pacote, alocar todas as estruturas necessárias e repassar o pacote para a camada de rede.

A estrutura *sk_buff* é uma das mais importantes de todo o subsistema de rede do Linux. É nela que os pacotes são armazenados com os cabeçalhos da camada de rede e dados do usuário. Ela é definida no arquivo *include/linux/skbuff.h* consistindo de um grande conjunto de variáveis tentando fornecer suporte a diversas operações (BENVENUTI, 2005).

Os cabeçalhos de camadas são referenciados por ponteiros. O Linux aloca uma estrutura *sk_buff*, normalmente sob o nome *skb*, onde ponteiros de *skb*, como *skb->transport_header*, fazem referência aos cabeçalhos. Para chegar aos dados do cabeçalho do protocolo, além do seu ponteiro são necessários outros: *skb->head*, *skb->data*, *skb->tail* e *skb->end*. Estes são ponteiros que representam limites de memória e dados na memória; *head* e o *end* apontam respectivamente para o início e o fim da memória alocada; *data* e o *tail* indicam o início e o fim da seção de dados (presente dentro do intervalo *head end*). Os ponteiros disponíveis em *sk_buff* para referência a camadas são: *mac_header* que aponta para a camada de enlace, *network_header* referenciando a camada de rede, e *transport_header* apontando para a camada de transporte. A Figura 19 apresenta a posição destes ponteiros.

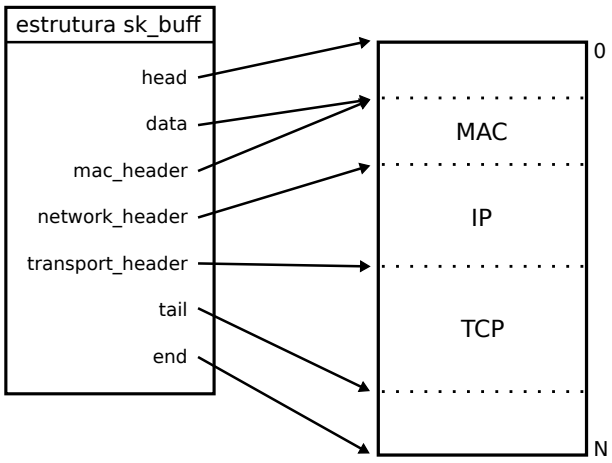

Figura 19: Posição dos principais ponteiros da estrutura *sk_buff*.

O valor dos ponteiros do parágrafo anterior depende da arquitetura do processador. Arquiteturas de 32 *bits* ou menos armazenam nos apontadores o endereço completo do local de memória. Processadores com mais de 32 *bits* utilizam um endereço curto (*offset*) para apontar, ou seja, quando se deseja obter o endereço completo deve-se somar ao seu valor o ponteiro *skb->head*.

Quando um pacote vai ser encaminhado o Linux não ajusta o ponteiro *skb->transport_header*, ficando este com o mesmo valor de *skb->network_header*. A fila *NGWA*, linha 5, precisa ter acesso ao cabeçalho de transporte quando o modo DPF está ativado. O ajuste do ponteiro é feito com o código presente na Figura 20, onde *ip_len* é o tamanho do cabeçalho IP.

```
#ifdef NET_SKBUFF_DATA_USES_OFFSET    /* arquitetura > 32 bits */
      skb->transport_header = skb->network_header + ip_len;
#else    /* arquitetura de 32 bits ou menos */
      skb->transport_header = skb->head + (skb->network_header - skb->head) + ip_len;
#endif
```

Figura 20: Código do ajuste do ponteiro *skb->transport_header* na fila *NGWA*.

4.1.2 DPF

A divisão por fluxos é uma característica do *NGWA* que pode ser ativada ou desativada. No Linux a DPF é por padrão desativada, podendo ser ativada através do carregamento do módulo *sch_ngwa* com o parâmetro *dpf=1*. Sua implementação, linhas 3 à 12, verifica o tipo do segmento TCP, pela checagem dos *bits* de controle do cabeçalho, realizando uma operação apropriada sobre uma lista duplamente encadeada.

Se o segmento for do tipo *SYN* e a conexão não tiver sido registrada anteriormente, ela é inserida na lista. Caso o segmento seja *FIN* ou *RST* a conexão é removida da lista [2]. Foi usada a estrutura *list_head*, declarada no arquivo *linux/list.h*, para representar a lista encadeada; a partir do *kernel* 2.1 foi construído este padrão para uso de listas ligadas, que além de simplificar seu uso fornece um conjunto de operações para manipulação de listas (LOVE, 2010). Para evitar registros duplicados na lista foi utilizado um conjunto de variáveis para identificar cada conexão, sendo eles: porta TCP de origem e destino, endereço IP de origem e destino, e identificação de protocolo. Antes de registrar uma nova conexão as variáveis são comparadas com os dados do novo pacote, sendo também feita uma comparação cruzada, comparando origem/destino da lista com origem/destino do pacote, por exemplo, é chegado se a porta de origem é igual a porta de origem ou de destino do novo pacote. A estrutura (conjunto de variáveis) para identificar a conexão ocupa 104 *bits* de memória; a Tabela 3 apresenta uma comparação do número de conexões com o uso de memória.

A linha 13 faz uma chamada a função *obter_bytes_fila*, é nela que ocorre a aplicação da DPF. A quantidade de *bytes* disponíveis da fila é dividida pelo número de conexões registradas.

[2] Recorda-se que se está considerando roteamento estático.

Tabela 3: Comparação do número de conexões com uso de memória.

Conexões	Memória
1	104 *bits*
10	125 *Bytes*
100	1.2 *kBytes*
1.000	12.5 *kBytes*
10.000	125 *kBytes*
100.000	1.2 *MBytes*

Na implementação no Linux as variáveis *aa* e *hs* se tornaram os parâmetros *dpf_aa* e *dpf_hs*.

4.1.3 CAMPO OPTIONS

Na última linha do pseudocódigo, a função *adicionar_ngwa_opt* é chamada para realizar a inserção/atualização do parâmetro *bytes_fila* no campo *options* do cabeçalho IP. O *options* presente no cabeçalho IP foi definido já na criação do protocolo IP pela RFC 791, sendo posicionado como o último campo do cabeçalho com um tamanho variável devendo ser um múltiplo de 32 *bits* (ou usar enchimento (*padding*)). Existem dois formatos possíveis: um simples octeto, ou um conjunto de octetos ordenados: primeiro octeto representa o tipo da opção, o segundo indica o tamanho da opção, e por fim os octetos de dados (POSTEL, 1981a). O *NGWA* foi implementado seguindo este último formato.

O Linux oferece suporte ao *options* do cabeçalho IP de uma maneira simples, tendo praticamente todo seu código no arquivo *net/ipv4/ip_options.c*. O uso do *options* é feito com a estrutura *ip_options* definida em *include/net/inet_sock.h*, suas principais variáveis são: *optlen*, indica o tamanho total em *bytes* das opções; *__pad2* é um ponteiro de uso genérico, podendo ser utilizado para novas opções; e *__data*, sendo um vetor para armazenar os dados de uma opção. As principais funções envolvidas na sua utilização são a *ip_options_build* e a *ip_options_compile*. A primeira é utilizada para escrever o *options* no cabeçalho IP, a estrutura *ip_options* é passada como parâmetro sendo gravado o conteúdo da variável *__data* no IP; *ip_options_compile* realiza a checagem das opções e as armazena em uma estrutura *ip_options* (WEHRLE et al., 2004) (HERBERT, 2004).

Foi criado um novo tipo de opção para ser utilizado pelo *NGWA*; o tipo deve ter o tamanho de oito *bits* e não estar em uso, foi escolhido o valor 33 (0010 0001). O *options* do *NGWA* é utilizado apenas a nível de *kernel*, não tendo interface para o usuário. A Figura 21 exibe o formato de uma opção *NGWA*. Ela é armazena em um vetor, onde cada índice tem o tamanho de um *byte*, tendo na posição inicial o valor 33, no índice um o tamanho do vetor, e sob o dois/três uma quantidade de *bytes*, limitada ao valor de 65536.

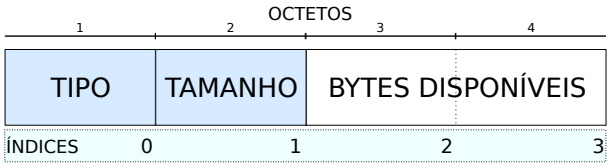

Figura 21: Estrutura do campo *options* do *NGWA*.

A criação/atualização do campo *options* é efetuada pela função *adicionar_ngwa_opt* que consiste em uma série de instruções, listadas a seguir:

- Alocação: Se o pacote passante não possuir o campo *options*, é necessário alocar um espaço de memória para seu armazenamento, pois o pacote recebido pela fila já está formado e armazenado linearmente na memória sob uma estrutura *sk_buff*. A tarefa de alocação e ajuste da estrutura *sk_buff* é realizada pela função (nova) *pskb_allocate_ip_options* localizada no arquivo *net/core/skbuff.c*. O tamanho em *bytes* (fornecido como parâmetro) é alocado por ela, após isso é preciso ajustar alguns ponteiros; *skb->transport_header*, *skb->tail*, e *skb->end* são incrementados pelo número de *bytes* requisitados. A Figura 22 exibe este processo.

- Preenchimento: Utilizando a função *kmalloc* uma estrutura do tipo *ip_options* é alocada. O vetor representado na Figura 21 é referenciado como parte da estrutura *ip_options* identificado pela variável *__data*. A informação da quantidade de *bytes* livres da fila do roteador tem 16 *bits* e é armazenada nos dois últimos octetos; para isto ser possível o valor de 16 *bits* é convertido em duas partes de oito *bits* — Equação 9.

$$\begin{aligned} ngwa &= bytes\ fila \\ parte1 &= ngwa\ \&\ 0xff \\ parte2 &= (ngwa >> 8)\ \&\ 0xff \end{aligned} \tag{9}$$

- Ajuste IP: O campo IHL do cabeçalho IP é utilizado para indicar a quantidade de blocos de 32 *bits* que o cabeçalho IP possuí, logo, quando é adicionado o campo *options* é necessário incrementar este valor. Para finalizar é necessário escrever a variável *__data* da estrutura *ip_options* no espaço alocado para o *options*, e isso é feito pela função *ip_options_build*.

4.1.4 UTILIZAÇÃO

O uso do módulo da fila *NGWA* segue a mesma forma que qualquer módulo do Linux. Alguns exemplos:

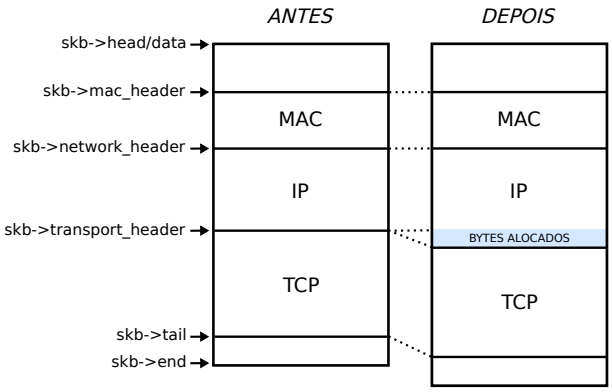

Figura 22: Alocação de *bytes* para o campo *options* na estrutura *sk_buff*.

- Carregar o módulo: **modprobe sch_ngwa**
  Realiza a leitura do módulo e sua carga para a memória. O modo DPF não é ativado por padrão.

- Ativar modo DPF: **modprobe sch_ngwa dpf=1**
  Também é possível ajustar o número médio de ACKs e o tamanho da cabeçalho pelos respectivos parâmetros **dpf_aa** e **dpf_hs**.

O Linux associa a cada interface de rede uma fila, por padrão é utilizado o modelo FIFO. Quando se quer utilizar a fila *NGWA* é preciso fazer um ajuste explícito: **tc qdisc add dev <int> handle 1:0 root ngwa**, onde <int> representa o nome da interface de rede.

4.2 TCP NGWA

O sistema *NGWA* traz informações da rede no campo *options* do cabeçalho IP. Esta informação é utilizada pelo TCP como um parâmetro indireto para a taxa de transmissão, pois o TCP *NGWA* do lado receptor irá utilizar a informação na computação do campo janela (*window*) do cabeçalho TCP — Equação 6, e o ACK recebido pelo transmissor é processado normalmente, porém contém uma janela mais adequada para a rede. O código para realizar este processo foi inserido diretamente no TCP do Linux, sendo dividido em entrada e saída TCP; tendo como sujeito o recebimento ou envio de pacotes, a divisão agrupa funções de cada um deles em um arquivo separado, definindo desta forma a entrada e saída TCP.

4.2.1 ENTRADA TCP

Um segmento TCP recebido é tratado na entrada, onde durante uma conexão todos os segmentos passam pela função *tcp_rcv_established*, localizada no arquivo *net/ipv4/tcp_input.c*,

os principais códigos para a entrada TCP foram inseridos nesta função.

Os códigos do *NGWA* somente são executados se a variável *sysctl_tcp_ngwa_on* estiver ativa; ela é controlada em tempo de execução a nível de usuário. Porém antes de chegar a este nível, na camada de rede é compilado o segmento recebido para o ajuste da estrutura *ip_options*, isto é feito pela rotina *ip_options_compile*. Voltando a camada de transporte, na chegada de um segmento é verificado a existência do campo *options* no cabeçalho IP, caso exista, é checado se a variável *opt->__pad2* foi ajusta.

Neste momento, a variável *opt->__pad2* é um ponteiro com a mesma estrutura vista na Figura 21. É verificado se a opção é do tipo *NGWA* (leitura da posição zero), em caso afirmativo, é realizada a leitura dos octetos. Sobre estas duas posições está localizada a informação desejada, como ela foi previamente codificada (16 bits em duas variáveis de 8 bits) é necessário realizar o processo reverso, sendo feito pela Equação 10. O resultado é armazenado em uma estrutura *tcp_sock* na variável *ngwa_ipopt*.

$$ngwa = parte2 << 8 \mid parte1 \tag{10}$$

4.2.2 SAÍDA TCP

A saída utiliza a informação extraída da entrada como parâmetro de cálculo da janela do cabeçalho TCP. Todo o código da saída foi inserido no arquivo *net/ipv4/tcp_output.c*, assim como na entrada TCP, os códigos são executados apenas pela ativação explícita do *NGWA*.

A informação do *NGWA* já foi salva previamente pela entrada TCP na estrutura *tcp_socket* sob a variável *ngwa_ipopt*, e agora ela é utilizada pela segunda parte da Equação 11 para obter uma nova janela; a equação foi posicionada dentro da função *tcp_select_window*. A suavização pode ser ativada em tempo de execução pela variável *sysctl_tcp_ngwa_mitigation*, e neste caso a primeira parte da equação é processada para atualizar a variável *ngwa_avg*; ela é igual a Equação 7, porém α tem peso 10 para evitar ponto flutuante, tornando o código do Linux mais adequado e simples. A variável *ngwa_avg* também foi incluída na estrutura *tcp_socket* e a constante α tem por padrão o valor três, podendo ser alterado a nível de usuário. A variável W_a é gerada pelo Linux.

$$ngwa\_avg = \frac{(10-\alpha)\cdot ngwa\_avg + \alpha \cdot ngwa\_ipopt}{10}$$
$$W_a = \begin{cases} min[W_{max}, W_a, ngwa\_ipopt], & \text{sem suavização.} \\ min[W_{max}, W_a, ngwa\_avg], & \text{com suavização.} \end{cases} \quad (11)$$

Na seção 4.1.3 foi descrito que quando o pacote não contiver o campo *options* no cabeçalho IP ele é adicionado. Naturalmente, com a adição de *bytes* no pacote ele terá um tamanho maior, podendo ultrapassar o tamanho máximo de pacote suportado pela rede. Para evitar que isso aconteça, a saída TCP também realiza uma diminuição do tamanho máximo do segmento (MSS) pelo tamanho da opção *NGWA*.

4.2.3 UTILIZAÇÃO

Diferentemente da fila, o TCP *NGWA* não pode ser carregado e descarregado da memória; ele é estático, e seu código foi inserido na pilha TCP/IP do Linux. Embora ele fique presente na memória, seu uso pode ser controlado, como segue:

- Ativar o TCP *NGWA*: **echo 1 > /proc/sys/net/ipv4/tcp_ngwa_on**
  Ativa o funcionamento do *NGWA* na pilha TCP/IP.
- Ativar a suavização: **echo 1 > /proc/sys/net/ipv4/tcp_ngwa_mitigation**
  Habilita a suavização. Antes de ativá-la é necessário ativar o *NGWA* com o comando anterior. Por padrão o α da suavização é três, pode-se trocar este valor, por exemplo, ajustar α para sete: **echo 7 > /proc/sys/net/ipv4/tcp_ngwa_mitigation_alpha**.

4.3 RESULTADOS

Para avaliar o desempenho da implementação no Linux foi montado um ambiente físico composto de computadores e cabos de rede. A ligação entre os dispositivos desta rede foi feita de forma direta, como pode ser observado na Figura 23. Todos os computadores da topologia tinham a mesma configuração de *hardware* e *software* detalhadas na Tabela 4.

Os testes foram realizados utilizando a topologia acima tendo quatro configurações de rede distintas no meio da rede (*backbone*). A velocidade do enlace central é de um Gbps, porém foram utilizadas as velocidades de 10 e 100 Mbps. No Linux para efetuar este ajuste foi usado a ferramenta *ethtool*. Em combinação com as velocidades foram utilizadas filas de 100 e 1000 pacotes.

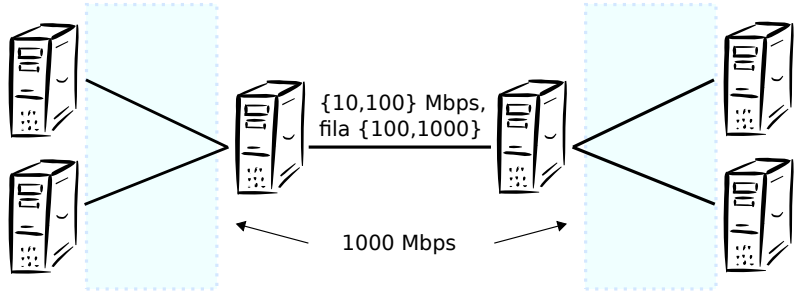

Figura 23: Topologia para testes.

Tabela 4: Configuração dos Computadores com Linux

Tópico	Descrição
Sistema Operacional (SO)	Debian GNU/Linux 6.0
Arquitetura do SO	i386
Linux	2.6.34 com patches *NGWA*
Processador	Core 2 Duo
Placa de Rede	Realtek RTL-8169 Gigabit Ethernet
Memória RAM	1024 MB

Utilizando as quatro configurações de rede citadas foram realizados experimentos com o *NGWA* (em quatro formas: normal, suavizado, DPF, e DPF suavizado), TCP New Reno, e TCP New Reno mais RED; em todos os casos foram obtidas 30 amostras. A geração de tráfego foi feita com a ferramenta *iperf*, pois é adequada para o caso e já foi usada em trabalhos similares, como em (LI et al., 2007) e (WEI; CAO, 2006); ela foi configurada para utilizar o protocolo TCP com janela de 64 *KBytes*, transmitindo dados constantemente em um sentido (cliente para servidor), informando a taxa de transmissão a cada meio segundo. Nos testes as conexões TCP foram estabelecidas do lado extremo esquerdo com o lado extremo direito da topologia, sendo coletada a quantidade de mega *bits* transmitidos.

Os experimentos foram realizados com um fluxo, e com dois fluxos simultâneos. Nos dois testes cada nó do lado esquerdo estabeleceu uma conexão TCP com um nó do lado direito, onde as fontes *on/off* estão transmitindo constantemente. Os dados obtidos foram analisados com base no método "*batch means*"(CHIEN, 1994), com 30 "*batches*", para a obtenção do intervalo de confiança de 95% presentes nos gráficos.

4.3.1 UM FLUXO

Este teste é simples, onde apenas um fluxo TCP é considerado. Foi estabelecida uma conexão de um computador do lado extremo esquerdo com um do extremo direito através do programa *iperf*. Cada experimento foi executado por dois minutos, pois após isso a rede atinge

um comportamento constante. Nos testes com o TCP New Reno foi utilizada uma fila FIFO, exceto quando o RED foi explicitamente configurado. Os parâmetros do RED foram ajustados conforme o tamanho da fila; o tamanho médio dos pacotes foi considerado em *1700 bytes*, logo para uma fila com tamanho de 100 pacotes: limite máximo = 170 *KBytes*, *minth* = 20 *KBytes*, *maxth* = 84 *KBytes*; com o tamanho de 1000 pacotes multiplicou-se por 10 os valores anteriores; para ambos os casos a probabilidade de descarte foi de *0,02* (WILLE et al., 2004).

O resultado do teste é visto na Figura 24, onde é exibida a taxa de transmissão em *bytes* por segundo. Nela pode-se observar que o desempenho do New Reno e do RED melhoraram significativamente com o crescimento do tamanho da fila, já os resultados do *NGWA* pouco mudaram. Nos quatro tipos de testes o resultado manteve-se: *NGWA* com suavização obtendo melhores resultados que os demais, sendo seguido pelo *NGWA* normal.

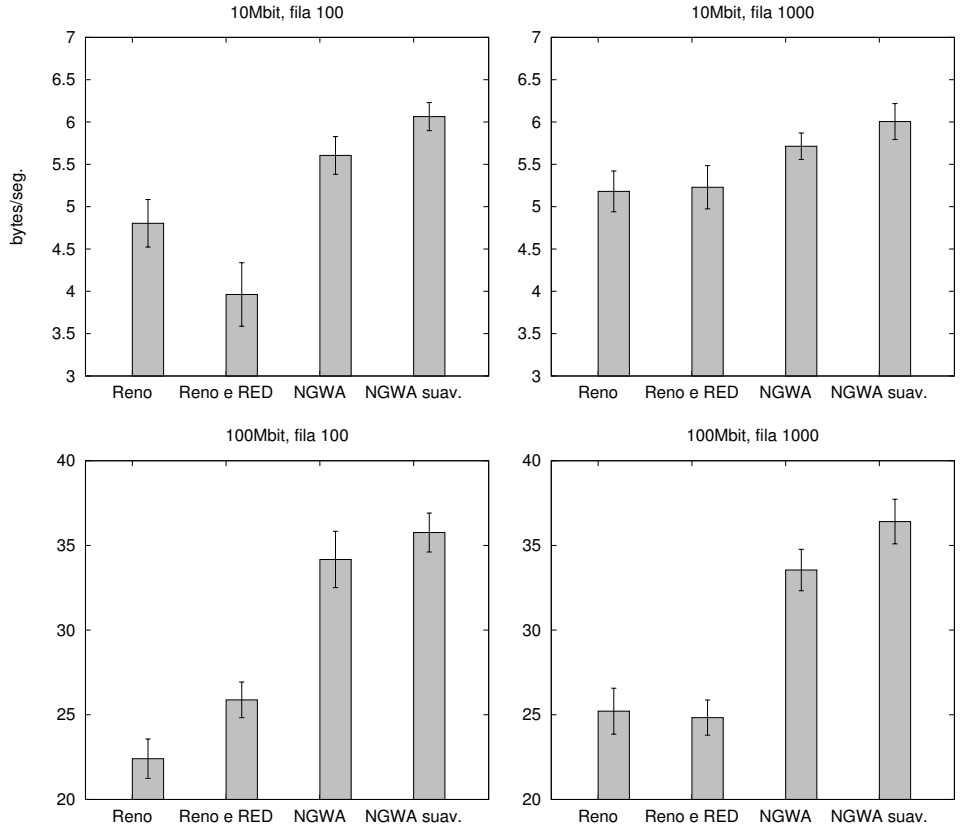

Figura 24: Taxa de transmissão de um fluxo no Linux.

4.3.2 DOIS FLUXOS

Neste experimento mais dois computadores foram utilizados, desta forma, dois computadores do lado esquerdo transmitem para dois do lado direito. O *iperf* foi executado nos quatro computadores, sendo coletados dois conjuntos de dados, um para o primeiro par

(fluxo 1) e outro para os outros dois computadores (fluxo 2). Considerando a topologia da Figura 23, onde os enlaces da esquerda e da direita da rede são de 1024 Mbps, tendo ao centro da rede um enlace de 10 Mbps, as transmissões estão sujeitas a um atraso de aproximadamente 100 segundos, e com dois fluxos simultâneos o atraso pode dobrar, impossibilitando um teste com o enlace central de 10 Mbps. Os demais detalhes da simulação são iguais aos da Seção anterior.

Neste segundo teste além de verificar o volume de dados é possível comparar a justiça (*fairness*) entre os fluxos. Analisando os gráficos da Figura 25, para a fila de 100 pacotes a divisão por fluxos funcionou adequadamente, além de obter um ótimo resultado para o *NGWA* com suavização. Com a fila em 1000 pacotes a justiça do *NGWA* diminuiu, porém ainda continuando mais alta que os demais métodos. A taxa de transmissão do *NGWA* em todas as suas formas foi maior que o New Reno e o RED em ambos os casos. Analisando o desempenho do New Reno e do RED, ou eles tiveram uma boa taxa de transmissão, ou a justiça foi razoável, nunca as duas juntas.

4.4 ANÁLISE

Não é possível comparar diretamente os resultados obtidos ao Capítulo 3 com aqueles apresentados neste capítulo, pois além de serem de natureza diferente (simulador e ambiente real), as topologias também são. Porém é aceitável comparar os resultados de cada ambiente, avaliando o *NGWA* em geral.

Considerando isto, analisando os resultados do *NGWA* implementado no simulador e no ambiente real, verificou-se que os resultados foram coerentes: no *NGWA* normal (sem suavização ou DPF) o volume de dados transmitido foi alto, porém ocorreu injustiça na rede; a DPF trouxe maior justiça para os fluxos; e a aplicação da suavização melhorou os resultados.

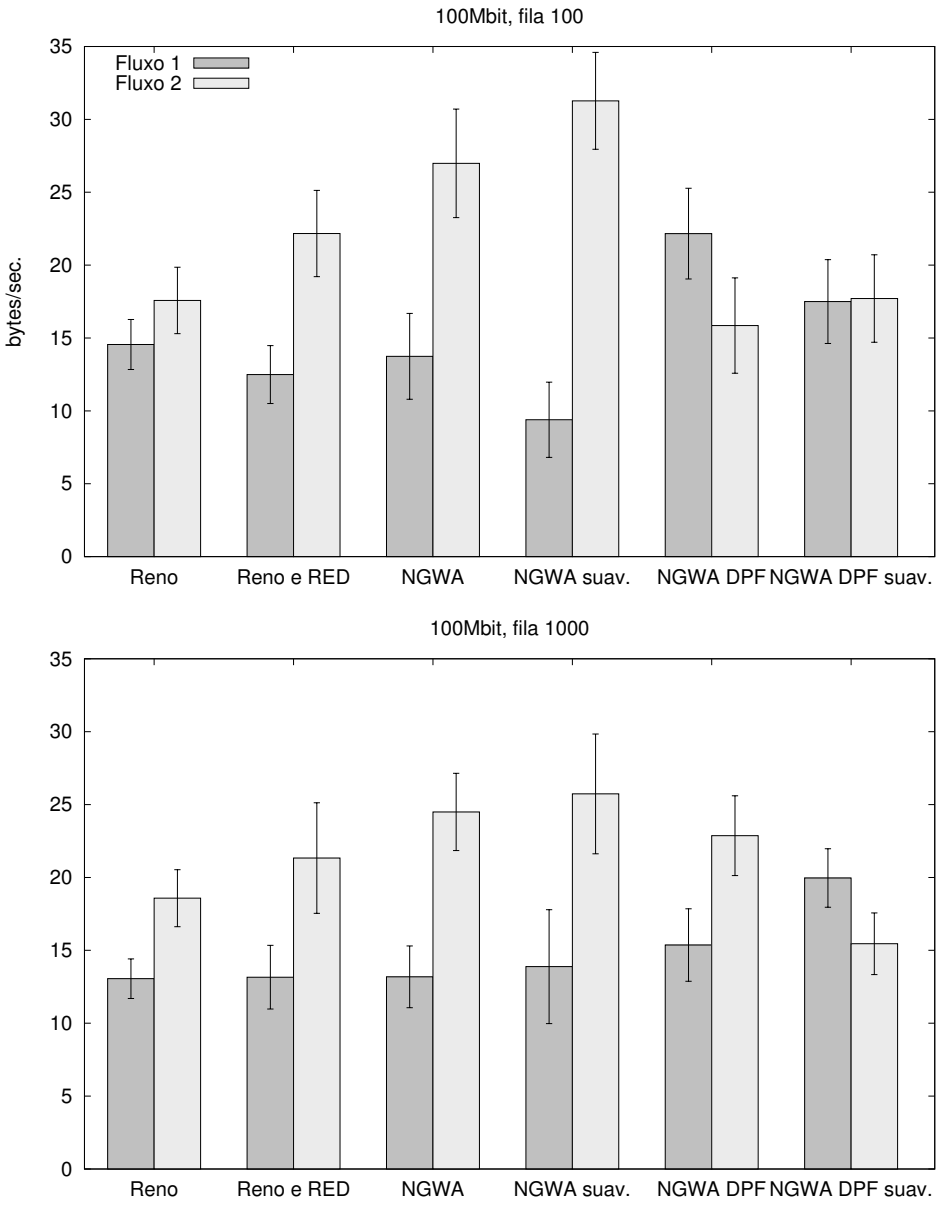

Figura 25: Taxa de transmissão de dois fluxos no Linux.

5 CONCLUSÃO

Neste trabalho foi apresentado um novo esquema para controle de congestionamento do protocolo TCP. O esquema denominado *new generalized window advertising (NGWA)* traz informações do uso de banda da rede para os pontos finais da conexão TCP. O *NGWA* foi subdividido em oito modos, cada qual com um conjunto de características diferentes. Foram realizados testes de taxa de transmissão e justiça (*fairness*) de uso de rede, variação da janela de transmissão, utilização de fila, e taxa de pacotes perdidos; todos os testes foram executados em uma rede operando com roteamento estático, e tendo dados gerados constantemente mediante uso de fontes *on-off*. Também foi feita a implementação do *NGWA* em um ambiente real (Linux), onde experimentos relativos a taxa de transmissão foram executados.

Os resultados foram comparados com o TCP New Reno, RED, e o TCP padrão. Nos testes realizados o *NGWA* apresenta uma operação mais estável e um bom desempenho. Comparado ao TCP New Reno e o TCP padrão, o *NGWA* obteve um desempenho superior em todos os testes, com destaque para o número de perdas, onde os modos 5/6/7/8 não registraram o recebimento de *triple dupacks* e perdas.

A fila dos roteadores com o uso do *NGWA* manteve-se estabilizada e com um alto nível de utilização. O emprego da suavização melhorou o desempenho dos modos 1 e 3. A divisão por fluxos (DPF) produziu resultados extremamente superiores ao demais modos.

Também foram realizados testes com a implementação do *NGWA* no Linux, onde foi registrado a taxa de transmissão. Pode-se observar que assim como no NS-3 a suavização tornou os resultados do *NGWA* melhores. E no geral o *NGWA* realizou uma maior taxa de transmissão, e na presença de dois fluxos a DPF foi eficaz, resultando em maior justiça.

Analisando os resultados do *NGWA* implementado no simulador e no ambiente real, verificou-se que os resultados foram coerentes: No *NGWA* normal (sem suavização ou DPF) a taxa de transmissão foi mais elevada, porém ocorreu injustiça na rede; a DPF trouxe maior justiça para os fluxos; e a aplicação da suavização melhorou os resultados.

5.1 TRABALHOS FUTUROS

Os estudos realizados indicaram que o esquema proposto é promissor. Entretanto, as simulações realizadas não permitiram avaliar todo o potencial e/ou possíveis deficiências do *NGWA*. Claramente melhorias na proposta são possíveis e algumas questões merecem uma melhor investigação.

- Um conceito fundamental do *NGWA* é a utilização da quantidade de *bytes* livres da fila dos roteadores como um parâmetro para o controle de congestionamento. É necessário estudar a eficácia do método ao tratar da contabilização das conexões ativas que fazem uso dos roteadores. Em uma rede dinâmica pode ocorrer processos de re-roteamento e/ou perda de fluxos devido a falhas. Neste caso, sugere-se que as estatísticas necessárias ocorram por "janela de tempo"de modo a assegurar uma melhor representação do estado das conexões.

- Analizar o desempenho da rede quando o *NGWA* compete com diferentes esquemas de controle de congestionamento, como o TCP New Reno, TCP CUBIC, e outros tipos de tráfego, como por exemplo, o UDP. Os modos 7 e 8 são os que menos utilizam recursos computacionais, pois não utilizam de controle de congestionamento. É importante dar ênfase nos testes para estes modos, e analizar a validade da sugestão de (JIANG et al., 2009), onde é sugerida a possibilidade de retirada total de mecanismos de controle de congestionamento do protocolo do TCP.

- Realização de testes em redes mais complexas, com maior número de nós e enlaces, fazendo com que existam rotas de diferentes RTTs. Sabe-se que quando existem fluxos que competem por recursos limitados, as conexões com RTTs mais elevados serão as mais penalizadas, reduzindo a justiça da rede. De modo geral, neste caso, as fontes reagem com atraso. Assim, é necessário verificar qual seria o impacto no desempenho do *NGWA* devido ao atraso no recebimento dos ACKs.

- Utilizar o ambiente implementado em Linux para realizar testes difíceis ou impossíveis de serem realizados via simulação no NS-3 como, por exemplo, usar velocidades de transmissão de 1 *Gbps*.

- Alteração do código TCP do Linux para ser possível o uso do TCP padrão, pois atualmente o controle de congestionamento não pode ser desacoplado do *kernel*, com isso os modos 7 e 8 do *NGWA* também poderiam ser utilizados nele.

REFERÊNCIAS

ALLMAN, M.; PAXSON, V. On estimating end-to-end network path properties. In: **SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication**. New York, NY, USA: ACM, 1999. p. 263–274. ISBN 1-58113-135-6.

BALAKRISHNAN, H.; SESHAN, S.; KATZ, R. H. Improving reliable transport and handoff performance in cellular wireless networks. **Wireless Networks**, ACM, Hingham, MA, USA, v. 1, n. 4, p. 469–481, 1995. ISSN 1022-0038.

BENVENUTI, C. **Understanding Linux network internals - guided tour to networking on Linux**. Gravenstein Highway North, Sebastopol, CA: O'Reilly, 2005. ISBN 978-0-596-00255-8.

BRADEN, B. et al. **RFC 2309: Recommendations on Queue Management and Congestion Avoidance in the Internet**. 1998. 16 p. Disponível em: <http://www.ietf.org/rfc/rfc2309.txt>. Acesso em: 12 set. 2011, 00:19.

BRAKMO, L.; PETERSON, L. TCP Vegas: end to end congestion avoidance on a global Internet. **Selected Areas in Communications**, IEEE Press, v. 13, n. 8, p. 1465–1480, 1995. ISSN 0733-8716.

CHIEN, C. Batch size selection for the batch means method. In: **Proceedings of the 26th conference on Winter simulation**. San Diego, CA, USA: Society for Computer Simulation International, 1994. (WSC '94), p. 345–352. ISBN 0-7803-2109-X.

COMER, D. **Internetworking with TCP/IP: principles, protocols, and architecture (vol. 1)**. New Jersey: Prentice Hall, 2000.

EWALD, N. L.; KEMP, A. H. Analytical Model of TCP NewReno through a CTMC. In: BRADLEY, J. T. (Ed.). **EPEW**. [S.l.]: Springer, 2009. v. 5652, p. 183–196. ISBN 978-3-642-02923-3.

FALL, K.; FLOYD, S. Simulation-based comparisons of Tahoe, Reno and SACK TCP. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 26, p. 5–21, jul. 1996. ISSN 0146-4833.

FLOYD, S. Connections with multiple congested gateways in packet-switched networks part 1: one-way traffic. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 21, n. 5, p. 30–47, 1991. ISSN 0146-4833.

FLOYD, S. TCP and explicit congestion notification. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 24, n. 5, p. 8–23, 1994. ISSN 0146-4833.

FLOYD, S.; JACOBSON, V. Random early detection gateways for congestion avoidance. **IEEE/ACM Transactions on Networking (TON)**, IEEE Press, Piscataway, NJ, USA, v. 1, n. 4, p. 397–413, 1993. ISSN 1063-6692.

GARCIA, M. et al. An experimental study of Snoop TCP performance over the IEEE 802.11b WLAN. In: **Wireless Personal Multimedia Communications, 2002. The 5th International Symposium on**. Honolulu, Hawaii, USA: IEEE Press, 2002. v. 3, p. 1068–1072. ISSN 1347-6890.

GERLA, M.; CIGNO, R. L.; WENG, W. Generalized Window Advertising for TCP Congestion Control. **European Transactions on Telecommunications**, v. 13, n. 6, p. 549–562, dez. 2002.

GERLA, M. et al. TCP Westwood: congestion window control using bandwidth estimation. In: **Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE**. San Antonio, USA: IEEE Press, 2001. v. 3, p. 1698–1702.

GERLA, M.; WENG, W.; CIGNO, R. Bandwidth feedback control of TCP and real time sources in the Internet. In: **Global Telecommunications Conference, 2000. GLOBECOM '00. IEEE**. San Francisco, USA: IEEE Press, 2000. v. 1, p. 561–565.

GERLA, M.; WENG, W.; CIGNO, R. L. BA-TCP: a bandwidth aware TCP for satellite networks. In: **Computer Communications and Networks, 1999. Proceedings. Eight International Conference on**. Boston, Massachusetts, USA: IEEE Press, 1999. p. 204–207.

GERLA, M.; WENG, W.; CIGNO, R. L. Enforcing fairness with explicit network feedback in the Internet. In: **Local and Metropolitan Area Networks, 1999. Selected Papers. 10th IEEE Workshop on**. Sydney, Australia: IEEE Press, 1999. p. 11–17.

HASEGAWA, G.; MURATA, M. TCP symbiosis: congestion control mechanisms of TCP based on Lotka-Volterra competition model. In: **Interperf '06: Proceedings from the 2006 workshop on Interdisciplinary systems approach in performance evaluation and design of computer & communications sytems**. New York, NY, USA: ACM, 2006. p. 11. ISBN 1-59593-503-7.

HERBERT, T. **The Linux TCP/IP stack: networking for embedded systems**. [S.l.]: Charles River Media, 2004. (Charles River Media programming series). ISBN 9781584502845.

JACOBSON, V. Congestion avoidance and control. In: **SIGCOMM '88: Symposium proceedings on Communications architectures and protocols**. New York, NY, USA: ACM, 1988. p. 314–329. ISBN 0-89791-279-9.

JACOBSON, V. **Modified TCP Congestion Avoidance Algorithm**. 1990. Disponível em: <ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail>. Acesso em: 16 ago. 2011, 15:06.

JAIN, M.; DOVROLIS, C. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. **IEEE/ACM Transactions on Networking (TON)**, IEEE Press, Piscataway, NJ, USA, v. 11, n. 4, p. 537–549, 2003. ISSN 1063-6692.

JIANG, S.; ZUO, Q.; WEI, G. Decoupling congestion control from TCP for multi-hop wireless networks: semi-TCP. In: **CHANTS '09: Proceedings of the 4th ACM workshop on Challenged networks**. New York, NY, USA: ACM, 2009. p. 27–34. ISBN 978-1-60558-741-7.

KALAMPOUKAS, L.; VARMA, A.; RAMAKRISHNAN, K. K. Explicit window adaptation: a method to enhance TCP performance. **IEEE/ACM Transactions on Networking**, v. 10, n. 3, p. 338–350, 2002. ISSN 1063-6692.

KODAMA, M.; HASEGAWA, G.; MURATA, M. Bandwidth-Based Congestion Control for TCP: Measurement Noise-Aware Parameter Settings and Self-Induced Oscillation. In: **Communications Workshops, 2009. ICC Workshops 2009. IEEE International Conference on**. Lenox, Massachusetts, USA: IEEE Press, 2009. p. 1–6.

KUROSE, J.; ROSS, K. **Redes de computadores e a Internet: uma abordagem top-down**. São Paulo, SP: Addison Wesley, 2010. ISBN 9788588639973.

LA, R.; ANANTHARAM, V. Charge-sensitive TCP and rate control in the Internet. In: **INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE**. Tel Aviv, Israel: IEEE Press, 2000. v. 3, p. 1166–1175.

LAI, Y.-C.; YAO, C.-L. The performance comparison between TCP Reno and TCP Vegas. In: **Parallel and Distributed Systems: Workshops, Seventh International Conference on, 2000**. Iwate, Japão: IEEE Press, 2000. p. 61–66.

LEUNG, K.-F.; YEUNG, K. G-Snoop: enhancing TCP performance over wireless networks. In: **Computers and Communications, 2004. Proceedings. ISCC 2004. Ninth International Symposium on**. Alexandria, Egypt: IEEE Press, 2004. v. 1, p. 545–550.

LI, Y.-T.; LEITH, D.; SHORTEN, R. Experimental Evaluation of TCP Protocols for High-Speed Networks. **Networking, IEEE/ACM Transactions on**, v. 15, n. 5, p. 1109–1122, out. 2007. ISSN 1063-6692.

Linux. The Linux Kernel. 2.6.34 2011. Disponível em: <http://www.kernel.org/pub/linux/kernel/ v2.6/linux-2.6.34.tar.bz2>. Acesso em: 18 nov. 2011, 12:53.

LOVE, R. **Linux Kernel Development**. 3(a). ed. Indianapolis, IN, USA: Addison Wesley, 2010. ISBN 0-672-32946-8.

LOW, S. H.; PETERSON, L.; WANG, L. Understanding TCP vegas: a duality model. In: **Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems**. New York, NY, USA: ACM, 2001. (SIGMETRICS '01), p. 226–235. ISBN 1-58113-334-0.

Lua. Lua: Licence. 3.11 2012. Disponível em: <http://www.lua.org/license.html>. Acesso em: 4 mar. 2012, 10:17.

MAN, C. L. T.; HASEGAWA, G.; MURATA, M. ImTCP: TCP with an inline measurement mechanism for available bandwidth. **Computer Communications**, Butterworth-Heinemann, Newton, MA, USA, v. 29, p. 1614–1626, jun. 2006. ISSN 0140-3664.

MASCOLO, S. Smith's predictor for congestion control in TCP Internet protocol. In: **American Control Conference, 1999. Proceedings of the 1999**. San Diego, California, USA: IEEE Press, 1999. v. 6, p. 4441–4445.

MATHIS, M.; HEFFNER, J.; REDDY, R. Web100: extended TCP instrumentation for research, education and diagnosis. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 33, n. 3, p. 69–79, 2003. ISSN 0146-4833.

MATHIS, M. et al. **RFC 2018: TCP Selective Acknowledgment Options**. 1996. Status: PROPOSED STANDARD. Disponível em: <ftp://ftp.internic.net/rfc/rfc2018.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc2018.txt>.

MO, J.; WALRAND, J. Fair end-to-end window-based congestion control. **IEEE/ACM Transactions on Networking (TON)**, IEEE Press, Piscataway, NJ, USA, v. 8, n. 5, p. 556–567, 2000. ISSN 1063-6692.

MOON, J.-C.; LEE, B. G. Rate-adaptive snoop: a TCP enhancement scheme over rate-controlled lossy links. **IEEE/ACM Transactions on Networking (TON)**, IEEE Press, Piscataway, NJ, USA, v. 14, n. 3, p. 603–615, 2006. ISSN 1063-6692.

MORRIS, R. TCP behavior with many flows. In: **International Conference on Network Protocols (ICNP '97)**. Atlanta, Georgia, USA: IEEE Press, 1997. p. 205–211. ISBN 0-8186-8061-X.

NS-2-LIMITATIONS. **NS-2 Limitations**. 2011. Disponível em: <http://www.isi.edu/nsnam/ns/ns-limitations.html>. Acesso em: 25 jan. 2011, 23:42.

NS-2-OUTDATED. **NS-2 outdated**. 2011. Disponível em: <http://www.nsnam.org/wiki/index.php/GSOC2011Projects>. Acesso em: 18 abr. 2011, 22:55.

NS-3. The Network Simulator. 3.11 2011. Disponível em: <http://www.nsnam.org/release/ns-allinone-3.11.tar.bz2>. Acesso em: 8 fev. 2011, 22:37.

NS-3-IPv6. **NS-3: IPv6 + TCP**. 2011. Disponível em: <http://mailman.isi.edu/pipermail/ns-developers/2010-December/008527.html>. Acesso em: 20 dez. 2011, 18:14.

POSTEL, J. B. **Internet Protocol**. 1981. RFC 791. Disponível em: <http://www.ietf.org/rfc/rfc791.txt>. Acesso em: 5 jan. 2011, 12:37.

POSTEL, J. B. **Transmission Control Protocol**. 1981. RFC 793. Disponível em: <http://www.ietf.org/rfc/rfc793.txt>. Acesso em: 12 jan. 2011, 19:19.

RAMAKRISHNAN, K.; FLOYD, S. **RFC 2481: A Proposal to add Explicit Congestion Notification (ECN) to IP**. 1999. Disponível em: <ftp://ftp.math.utah.edu/pub/rfc/rfc2481.txt>. Acesso em: 15 jan. 2011, 21:07.

RAMAKRISHNAN, K.; FLOYD, S.; BLACK, D. **RFC 3168: The addition of Explicit Congestion Notification (ECN) to IP**. 2001.

STEVENS, R. W. **TCP/IP Illustrated, Volume 1: The Protocols**. Indianapolis, IN, USA: Addison Wesley, 1994. ISBN 0-201-63346-9.

STEVENS, R. W. **RFC 2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms**. 1997.

SUN, F.; LI, V.; LIEW, S. Design of SNACK mechanism for wireless TCP with new snoop. In: **Wireless Communications and Networking Conference, 2004. WCNC. 2004 IEEE**. Atlanta, Georgia, USA: IEEE Press, 2004. v. 2, p. 1051–1056. ISSN 1525-3511.

TANENBAUM, A. **Redes de computadores**. Rio de Janeiro: Editora campus, 1997. ISBN 85-352-0157-2.

THIRUCHELVI, G.; RAJA, J. A Survey On Active Queue Management Mechanisms. **International Journal of Computer Science and Network Security (IJCSNS)**, v. 8, p. 130–145, 2008. ISSN 1738-7906.

VANGALA, S.; LABRADOR, M. The TCP SACK-aware snoop protocol for TCP over wireless networks. In: **Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th**. Orlando, Florida, USA: IEEE Press, 2003. v. 4, p. 2624–2628. ISSN 1090-3038.

WEHRLE, K. et al. **The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel**. Gravenstein Highway North, Sebastopol, CA: Prentice Hall, 2004. ISBN 0-13-177720-3.

WEI, D. X.; CAO, P. NS-2 TCP-Linux: an NS-2 TCP implementation with congestion control algorithms from Linux. In: **Proceeding from the 2006 workshop on ns-2: the IP network simulator**. New York, NY, USA: ACM, 2006. (WNS2 '06). ISBN 1-59593-508-8.

WEIGLE, E.; FENG, W. chun. Dynamic right-sizing: a simulation study. In: **Computer Communications and Networks, 2001. Proceedings. Tenth International Conference on**. St. Thomas, Virgin Islands, USA: IEEE Press, 2001. p. 152–158.

WEINGARTNER, E.; LEHN, H. vom; WEHRLE, K. A Performance Comparison of Recent Network Simulators. In: **Communications, 2009. ICC '09. IEEE International Conference on**. Lenox, Massachusetts, USA: IEEE Press, 2009. p. 1–5. ISSN 1938-1883.

WILLE, E. C. G. et al. Design and Analysis of IP Networks with End-to-End QoS Guarantees. In: **XXI Simpósio Brasileiro de Telecomunicações**. Belém-Pará, Brasil: Sociedade Brasileira de Telecomunicações, 2004.

YANG, Y.; LAM, S. General AIMD congestion control. In: **Network Protocols, 2000. Proceedings. 2000 International Conference on**. Osaka, Japão: IEEE Press, 2000. p. 187–198.

ZHANG, Y.; FENG, G. A new method to improve the TCP performance in wireless cellular networks. In: **Communications, Circuits and Systems, 2009. ICCCAS 2009. International Conference on**. Milpitas, California, USA: IEEE Press, 2009. p. 246–250.

ZHANG, Y.; HU, J.; FENG, G. SNOOP-based TCP Enhancements with FDA in wireless cellular networks: A comparative study. In: **Communications, Circuits and Systems, 2008. ICCCAS 2008. International Conference on**. Fujian Province, China: IEEE Press, 2008. p. 181–185.

ZHOU, K.; YEUNG, K.; LI, V. On Performance Modeling of TCP New-Reno. In: **Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE**. Washington, DC, USA: IEEE Press, 2007. p. 2650–2654.

ANEXO A – CÓDIGO FONTE DO *NGWA* PARA LINUX

Módulo:

```
/*
 * net/sched/sch_ngwa.c New Generalized Window Advertising
 *
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  as published by the Free Software Foundation; either version
 *  2 of the License, or (at your option) any later version.
 *
 * Author: Marcos Talau, <talau@users.sourceforge.net>
 *
 * UPDATES: The latest version of this source code can be obtained by the above address.
 */
#include <linux/module.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/types.h>
#include <linux/string.h>
#include <linux/errno.h>
#include <linux/skbuff.h>
#include <linux/rtnetlink.h>
#include <linux/bitops.h>
#include <linux/list.h>
#include <net/pkt_sched.h>
#include <net/ip.h>
#include <net/tcp.h>
#include <net/udp.h>
#include <asm/byteorder.h>

/* active DPF? */
static int dpf = 0;
module_param (dpf, bool, S_IRUGO);
MODULE_PARM_DESC (dpf, "true to use NGWA DPF at queue");
static int dpf_aa = 10;
module_param (dpf_aa, int, S_IRUGO);
MODULE_PARM_DESC (dpf_aa, "NGWA DPF aa (average ack)");
static int dpf_hs = 20;
module_param (dpf_hs, int, S_IRUGO);
MODULE_PARM_DESC (dpf_hs, "NGWA DPF hs (header size)");
```

```
39
40 extern int sch_total_pkt_size;
41 extern int sch_total_pkt;
42
43 struct ngwa_tcp_connection {
44         __be32 saddr;
45         __be32 daddr;
46         __be16 sport;
47         __be16 dport;
48         __u8 protocol;
49         struct list_head list;
50 };
51 struct ngwa_tcp_connection *ngwa_tcp_conn;
52 static LIST_HEAD(ngwa_tcp_conn_list);
53
54 struct ngwa_sched_data
55 {
56         u32 limit;
57         struct Qdisc  *q;
58 };
59
60 // used only for dpf param; the number of active connections between queue
61 static int dpf_conn = 0;
62
63 // return average queue len free in bytes
64 static u32 ngwa_get_available_bandwidth(struct Qdisc *sch)
65 {
66         struct ngwa_sched_data *q = qdisc_priv(sch);
67         u32 mean_pkt_size;
68         u32 bytes_free;
69
70         // with out (u32) the variable bstats.bytes use 64 bits,
71         // so in 32 bits compilation the error '__udivdi3' will appear
72         mean_pkt_size = (u32) sch_total_pkt_size / sch_total_pkt;
73         bytes_free = (q->limit - sch->q.qlen) * mean_pkt_size;
74
75         if (dpf && dpf_conn && (bytes_free > dpf_conn)) {
76                 printk(KERN_DEBUG "(ngwa_get_available_bandwidth) (dpf) total
                        bytes_free = %d\n", bytes_free);
77                 bytes_free /= dpf_conn;
78                 printk(KERN_DEBUG "(ngwa_get_available_bandwidth) (dpf) with %d conn
                        bytes_free = %d\n", dpf_conn, bytes_free);
79                 bytes_free = bytes_free - dpf_aa * (dpf_hs * dpf_conn);
80                 printk(KERN_DEBUG "(ngwa_get_available_bandwidth) (dpf) with %d conn
                        bytes_free (with aa=%d and hs=%d) = %d\n", dpf_conn, dpf_aa, dpf_hs,
                        bytes_free);
81         }
82
83         return bytes_free;
84 }
85
```

```
86  static u32 ngwa_get_ngwa_pkt(struct sk_buff *skb)
87  {
88          struct iphdr *iph;
89          struct ip_options *opt;
90          // will point to ip->opt real data
91          unsigned char *sptr;
92
93          iph = ip_hdr(skb);
94          opt = &(IPCB(skb)->opt);
95
96          if (opt && opt->optlen) {
97                  // ngwa are in __pad2 pointer
98                  if (opt->__pad2) {
99                          sptr = skb_network_header(skb);
100
101                         if (sptr[opt->__pad2] == IPOPT_NGWA) {
102                                 printk(KERN_DEBUG "(ngwa_get_ngwa_pkt) ngwa in
                                        received skb had %d\n", (u16) ((u16)
                                        sptr[opt->__pad2+3] << 8) | (u16)
                                        sptr[opt->__pad2+2]);
103                                 // 8 bits from __pad2+2 and 8 too from __pad2+3 = 16 bits
104                                 return (u16) ((u16) sptr[opt->__pad2+3] << 8) | (u16)
                                        sptr[opt->__pad2+2];
105                         }
106                 }
107         }
108         // if skb do not have ngwa already
109         return 0;
110 }
111
112 static void ngwa_update_pkt(struct Qdisc *sch, struct sk_buff *skb, u32 ngwa_bw)
113 {
114         // optlen in bytes
115         u8 optlen = NGWA_OLEN;
116         struct iphdr *iph = ip_hdr(skb);
117
118         __be32 daddr = 0;
119         struct ip_options *old_opt;
120         struct ip_options *new_opt;
121         // real because it have 16 bits
122         u16 real_bw;
123         // we will alloc ipopt? if ip already have ip->opt we don't
124         u8 alloc_ipopt = 0;
125
126         real_bw = min(ngwa_bw, MAX_TCP_WINDOW);
127
128         // check if already have ipopt
129         old_opt = &(IPCB(skb)->opt);
130         if (!(old_opt && old_opt->optlen)) {
131                 alloc_ipopt = 1;
132
```

```
133                     iph = ip_hdr(skb);
134                     if (iph->ihl != 5)
135                             printk(KERN_EMERG "hmm unexpected ihl len. FIXME\n");
136
137                     pskb_allocate_ip_options(skb, optlen, GFP_ATOMIC);
138             }
139
140             iph = ip_hdr(skb);
141
142             new_opt = kmalloc(sizeof (struct ip_options) + 40, GFP_KERNEL);
143             memset(new_opt, 0, sizeof(struct ip_options) + 40);
144             new_opt->__data[0] = IPOPT_NGWA; // type of option
145             new_opt->__data[1] = optlen; // optlen
146             // here [2]/[3] we split real_bw into two parts with 8 bits
147             // [2] = right part, [3] = left part
148             new_opt->__data[2] = real_bw & 0xff;
149             new_opt->__data[3] = (real_bw >> 8) & 0xff;
150             new_opt->__data[4] = IPOPT_NOOP;
151             new_opt->__data[5] = IPOPT_NOOP;
152             new_opt->__data[6] = IPOPT_NOOP;
153             new_opt->__data[7] = IPOPT_END;
154             new_opt->optlen = optlen;
155             if (alloc_ipopt)
156                     iph->ihl += new_opt->optlen>>2;
157
158             ip_options_build(skb, new_opt, daddr, NULL, 0);
159             iph = ip_hdr(skb);
160
161             if (alloc_ipopt) {
162                     iph->tot_len += htons(optlen);
163                     skb->len += optlen;
164             }
165
166             // add checksum
167             ip_send_check(iph);
168             atomic_set(&skb_shinfo(skb)->dataref, 1);
169
170             old_opt = &(IPCB(skb)->opt);
171             if (!(old_opt && old_opt->optlen)) {
172                     printk(KERN_EMERG "Oh shit! ngwa do not set options\n");
173             }
174
175             if (skb->len > 1518)
176                     printk(KERN_EMERG "NGWA problem. Packet (skb) size is too large. hint:
                            Check if MSS in TCP sides are with - 8?\n");
177
178             printk(KERN_DEBUG "(ngwa_update_pkt) updated to %d\n", real_bw);
179     }
180
181     static void ngwa_add_conn(struct iphdr *iph, struct tcphdr *tcph)
182     {
```

```
183         struct ngwa_tcp_connection *new;
184
185         new = kmalloc(sizeof(*new), GFP_KERNEL);
186         new->saddr = iph->saddr;
187         new->daddr = iph->daddr;
188         new->sport = tcph->source;
189         new->dport = tcph->dest;
190         new->protocol = iph->protocol;
191         list_add(&new->list, &ngwa_tcp_conn_list);
192         dpf_conn++;
193
194         printk(KERN_EMERG "(ngwa_add_conn) Now number of connections is %d.\n",
                dpf_conn);
195 }
196
197 static struct ngwa_tcp_connection * ngwa_find_conn(struct iphdr *iph, struct tcphdr *tcph)
198 {
199         struct ngwa_tcp_connection *conn;
200
201         list_for_each_entry(conn, &ngwa_tcp_conn_list, list) {
202                 if ( (iph->saddr == conn->saddr) && (iph->daddr == conn->daddr)
203                         && (tcph->source == conn->sport) && (tcph->dest ==
                            conn->dport)
204                         && (iph->protocol == conn->protocol))
205                         return conn;
206                 // two side check
207                 if ( (iph->saddr == conn->daddr) && (iph->daddr == conn->saddr)
208                         && (tcph->source == conn->dport) && (tcph->dest ==
                            conn->sport)
209                         && (iph->protocol == conn->protocol))
210                         return conn;
211         }
212
213         return NULL;
214 }
215
216 static void ngwa_del_conn(struct ngwa_tcp_connection *conn)
217 {
218         if (likely(conn)) {
219                 printk(KERN_DEBUG "(ngwa_del_conn) conn will be deleted\n");
220                 list_del(&conn->list);
221         } else {
222                 printk(KERN_DEBUG "----- conn NOT will be deleted\n");
223         }
224
225         if (dpf_conn == 0)
226                 return;
227         dpf_conn--;
228         printk(KERN_EMERG "(ngwa_del_conn) Now number of connections is %d.\n",
                dpf_conn);
229 }
```

```
230
231 // main ngwa cicle in queue
232 static void ngwa_process(struct Qdisc *sch, struct sk_buff *skb)
233 {
234         struct iphdr *iph = ip_hdr(skb);
235         struct tcphdr *tcph;
236         u32 ngwa_local_bw;
237         u32 ngwa_from_pkt;
238         struct ip_options *opt;
239         u8 ip_len = 20;
240
241         // ngwa will apply only in TCP protocol
242         if ((iph->frag_off & htons(IP_OFFSET) ? 0 : iph->protocol) != IPPROTO_TCP)
243                 return;
244
245         if (dpf) {
246                 // check if already have ipopt
247                 opt = &(IPCB(skb)->opt);
248
249                 if (opt && opt->optlen)
250                         ip_len += NGWA_OLEN;
251
252         // always keep transport_header correct (when forward a packet transport_header
                will not be adjusted)
253 #ifdef NET_SKBUFF_DATA_USES_OFFSET
254                 skb->transport_header = skb->network_header + ip_len;
255 #else
256                 skb->transport_header = skb->head + (skb->network_header -
                    skb->head) + ip_len;
257 #endif
258                 tcph = tcp_hdr(skb);
259
260                 printk(KERN_DEBUG "(ngwa_process)(dpf) DPF is on ;; active tcp conn =
                    %d\n", dpf_conn);
261                 printk(KERN_DEBUG "(ngwa_process)(dpf) out-int %s with ip pkt from:
                    %pI4 to: %pI4 ;; flags: fin %d, ack %d, psh %d, syn %d\n",
                    skb->dev->name, &iph->saddr, &iph->daddr, tcph->fin, tcph->ack,
                    tcph->psh, tcph->syn);
262                 // DPF part
263                 // here only for SYN+ACK
264                 if (tcph->syn) {
265                         printk(KERN_DEBUG "(ngwa_process)(dpf) syn/? received ;; (%pI4
                            => %pI4) flags: fin %d, ack %d, psh %d, syn %d\n",
                            &iph->saddr, &iph->daddr, tcph->fin, tcph->ack, tcph->psh,
                            tcph->syn);
266                         if (! ngwa_find_conn(iph, tcph)) {
267                                 printk(KERN_DEBUG "(ngwa_process)(dpf) new conn will
                                    be registred\n");
268                                 ngwa_add_conn(iph, tcph);
269                         }
270                 } else if (tcph->fin || tcph->rst) {
```

```
271                             printk(KERN_DEBUG "(ngwa_process)(dpf) fin or rst received\n");
272                             ngwa_del_conn(ngwa_find_conn(iph, tcph));
273                     }
274             }
275
276             ngwa_local_bw = ngwa_get_available_bandwidth(sch);
277             ngwa_from_pkt = ngwa_get_ngwa_pkt(skb);
278
279             printk(KERN_DEBUG "(ngwa_process) ngwa_local %d; ngwa_from_pkt %d\n",
                    ngwa_local_bw, ngwa_from_pkt);
280
281             if ((ngwa_local_bw < ngwa_from_pkt) || (ngwa_from_pkt == 0)) {
282                     printk(KERN_DEBUG "(ngwa_process) I will update pkt due less bw.\n");
283                     if (likely(ngwa_local_bw))
284                             ngwa_update_pkt(sch, skb, ngwa_local_bw);
285             }
286     }
287
288     // only for debug process
289     static void show_info(struct sk_buff *skb)
290     {
291             struct iphdr *iph = ip_hdr(skb);
292
293             printk(KERN_DEBUG "[sch_ngwa](SHOWINFO) [int %s]; ip version %d; from: %pI4
                    to: %pI4\n", skb->dev->name, iph->version, &iph->saddr, &iph->daddr);
294     }
295
296     static int ngwa_enqueue(struct sk_buff *skb, struct Qdisc* sch)
297     {
298             struct ngwa_sched_data *q = qdisc_priv(sch);
299             int ret;
300
301             ret = qdisc_enqueue(skb, q->q);
302             if (ret != NET_XMIT_SUCCESS) {
303                     if (net_xmit_drop_count(ret))
304                             sch->qstats.drops++;
305                     return ret;
306             }
307
308             sch->bstats.bytes += qdisc_pkt_len(skb);
309             sch->bstats.packets++;
310             sch->q.qlen++;
311
312             return NET_XMIT_SUCCESS;
313     }
314
315     static struct sk_buff * ngwa_dequeue(struct Qdisc* sch)
316     {
317             struct sk_buff *skb;
318             struct ngwa_sched_data *q = qdisc_priv(sch);
319
```

```
320         skb = q->q->ops->dequeue(q->q);
321         if (skb == NULL)
322                 return NULL;
323
324         sch_total_pkt_size += qdisc_pkt_len(skb);
325         if (sch_total_pkt == 0)
326                 sch_total_pkt = 1;
327
328         sch->q.qlen--;
329
330         if (skb)
331                 ngwa_process(sch, skb);
332
333         return skb;
334 }
335
336 static int ngwa_init(struct Qdisc *sch, struct nlattr *opt)
337 {
338         struct ngwa_sched_data *q = qdisc_priv(sch);
339         u32 limit = qdisc_dev(sch)->tx_queue_len ? : 1;
340
341         q->limit = limit;
342         sch->q.qlen = 0;
343         sch->bstats.bytes = 0;
344         sch->bstats.packets = 0;
345
346         if (dpf)
347                 printk(KERN_EMERG "(ngwa_init) DPF is active.\n");
348
349         printk(KERN_EMERG "(ngwa_init) sch_total_pkt_size restarted.\n");
350         sch_total_pkt_size = 0;
351         sch_total_pkt = 0;
352
353         q->q = qdisc_create_dflt(qdisc_dev(sch), sch->dev_queue,
354                                  &pfifo_qdisc_ops, sch->handle);
355         if (q->q == NULL)
356                 q->q = &noop_qdisc;
357
358         ngwa_tcp_conn = kmalloc(sizeof(*ngwa_tcp_conn), GFP_KERNEL);
359         INIT_LIST_HEAD(&ngwa_tcp_conn->list);
360
361         return 0;
362 }
363
364 static int ngwa_dump(struct Qdisc *sch, struct sk_buff *skb)
365 {
366         struct ngwa_sched_data *q = qdisc_priv(sch);
367         struct tc_fifo_qopt opt = { .limit = q->limit };
368
369         NLA_PUT(skb, TCA_OPTIONS, sizeof(opt), &opt);
370         return skb->len;
```

```
371
372 nla_put_failure:
373         return −1;
374 }
375
376 static unsigned int ngwa_drop(struct Qdisc ∗sch)
377 {
378         struct ngwa_sched_data ∗q = qdisc_priv(sch);
379         unsigned int len;
380
381         if (q−>q−>ops−>drop == NULL)
382                 return 0;
383
384         len = q−>q−>ops−>drop(q−>q);
385         if (len)
386                 sch−>q.qlen−−;
387
388         return len;
389 }
390
391 static struct sk_buff ∗ngwa_peek(struct Qdisc ∗sch)
392 {
393         struct ngwa_sched_data ∗q = qdisc_priv(sch);
394
395         return q−>q−>ops−>peek(q−>q);
396 }
397
398 static void ngwa_reset(struct Qdisc ∗sch)
399 {
400         struct ngwa_sched_data ∗q = qdisc_priv(sch);
401
402         qdisc_reset(q−>q);
403         sch−>q.qlen = 0;
404         sch−>bstats.bytes = 0;
405         sch−>bstats.packets = 0;
406 }
407
408 static int ngwa_graft(struct Qdisc ∗sch, unsigned long arg,
409                       struct Qdisc ∗new, struct Qdisc ∗∗old)
410 {
411         struct ngwa_sched_data ∗q = qdisc_priv(sch);
412
413         if (new == NULL) {
414                 new = qdisc_create_dflt(qdisc_dev(sch), sch−>dev_queue,
415                                         &pfifo_qdisc_ops,
416                                         sch−>handle);
417                 if (new == NULL)
418                         new = &noop_qdisc;
419         }
420
421         sch_tree_lock(sch);
```

```
422             *old = q->q;
423             q->q = new;
424             qdisc_tree_decrease_qlen(*old, (*old)->q.qlen);
425             qdisc_reset(*old);
426             sch_tree_unlock(sch);
427
428             return 0;
429     }
430
431     static struct Qdisc *ngwa_leaf(struct Qdisc *sch, unsigned long arg)
432     {
433             struct ngwa_sched_data *q = qdisc_priv(sch);
434             return q->q;
435     }
436
437     static unsigned long ngwa_get(struct Qdisc *sch, u32 classid)
438     {
439             return TC_H_MIN(classid) + 1;
440     }
441
442     static void ngwa_walk(struct Qdisc *sch, struct qdisc_walker *walker)
443     {
444             if (!walker->stop) {
445                     if (walker->count >= walker->skip)
446                             if (walker->fn(sch, 1, walker) < 0) {
447                                     walker->stop = 1;
448                                     return;
449                             }
450                     walker->count++;
451             }
452     }
453
454     static void ngwa_put(struct Qdisc *sch, unsigned long arg)
455     {
456             return;
457     }
458
459     static const struct Qdisc_class_ops ngwa_class_ops = {
460             .graft  = ngwa_graft,
461             .leaf  = ngwa_leaf,
462             .get  = ngwa_get,
463             .put  = ngwa_put,
464             .walk  = ngwa_walk,
465     };
466
467     static struct Qdisc_ops ngwa_qdisc_ops __read_mostly = {
468             .id  = "ngwa",
469             .priv_size = sizeof(struct ngwa_sched_data),
470             .cl_ops  = &ngwa_class_ops,
471             .enqueue = ngwa_enqueue,
472             .dequeue = ngwa_dequeue,
```

```
473             .peek  = ngwa_peek,
474             .drop  = ngwa_drop,
475             .init  = ngwa_init,
476             .reset  = ngwa_reset,
477             .change  = ngwa_init, // it can change
478             .dump  = ngwa_dump,
479             .owner  = THIS_MODULE,
480     };
481
482     static int __init ngwa_module_init(void)
483     {
484             return register_qdisc(&ngwa_qdisc_ops);
485     }
486
487     static void __exit ngwa_module_exit(void)
488     {
489             unregister_qdisc(&ngwa_qdisc_ops);
490     }
491
492     module_init(ngwa_module_init)
493     module_exit(ngwa_module_exit)
494
495     MODULE_LICENSE("GPL");
```

Patches:

```
1   diff --git a/include/linux/ip.h b/include/linux/ip.h
2   index bd0a2a8..63f5a6c 100644
3   --- a/include/linux/ip.h
4   +++ b/include/linux/ip.h
5   @@ -62,6 +62,10 @@
6    #define IPOPT_SID (8 |IPOPT_CONTROL|IPOPT_COPY)
7    #define IPOPT_SSRR (9 |IPOPT_CONTROL|IPOPT_COPY)
8    #define IPOPT_RA (20|IPOPT_CONTROL|IPOPT_COPY)
9   +#define IPOPT_NGWA 33
10  +
11  +// size in bytes of NGWA option (will be added to ip->opt)
12  +#define NGWA_OLEN 8
13
14   #define IPVERSION 4
15   #define MAXTTL  255
16  diff --git a/include/linux/skbuff.h b/include/linux/skbuff.h
17  index 124f90c..a7be950 100644
18  --- a/include/linux/skbuff.h
19  +++ b/include/linux/skbuff.h
20  @@ -452,6 +452,9 @@ extern struct sk_buff *skb_copy(const struct sk_buff *skb,
21                                          gfp_t priority);
22   extern struct sk_buff *pskb_copy(struct sk_buff *skb,
23                                           gfp_t gfp_mask);
24  +extern int pskb_allocate_ip_options(struct sk_buff *skb,
```

```
+    u8 ip_opt_size,
+    gfp_t gfp_mask);
 extern int pskb_expand_head(struct sk_buff *skb,
                             int nhead, int ntail,
                             gfp_t gfp_mask);
diff --git a/include/linux/tcp.h b/include/linux/tcp.h
index a778ee0..e3fd381 100644
--- a/include/linux/tcp.h
+++ b/include/linux/tcp.h
@@ -21,6 +21,7 @@
 #include <asm/byteorder.h>
 #include <linux/socket.h>

+// the tcp header
 struct tcphdr {
         __be16 source;
         __be16 dest;
@@ -378,6 +379,12 @@ struct tcp_sock {
         u32 snd_cwnd_used;
         u32 snd_cwnd_stamp;

+ // NGWA field from ip->options
+ // u32 to be compatible with min()
+ u32 ngwa_ipopt;
+ // used only to NGWA mitigation
+ u32 ngwa_avg;
+
         u32 rcv_wnd; /* Current receiver window */
         u32 write_seq; /* Tail(+1) of data held in tcp send buffer */
         u32 pushed_seq; /* Last pushed seq, required to talk to windows */
diff --git a/include/net/tcp.h b/include/net/tcp.h
index aa04b9a..f17a9c4 100644
--- a/include/net/tcp.h
+++ b/include/net/tcp.h
@@ -246,6 +246,9 @@ extern int sysctl_tcp_max_ssthresh;
 extern int sysctl_tcp_cookie_size;
 extern int sysctl_tcp_thin_linear_timeouts;
 extern int sysctl_tcp_thin_dupack;
+extern int sysctl_tcp_ngwa_on;
+extern int sysctl_tcp_ngwa_mitigation;
+extern int sysctl_tcp_ngwa_mitigation_alpha;

 extern atomic_t tcp_memory_allocated;
 extern struct percpu_counter tcp_sockets_allocated;
diff --git a/net/core/skbuff.c b/net/core/skbuff.c
index 93c4e06..059f00b 100644
--- a/net/core/skbuff.c
+++ b/net/core/skbuff.c
@@ -781,6 +782,101 @@ out:
 }
 EXPORT_SYMBOL(pskb_copy);
```

```
76
77 +// some notes:
78 +// * we mean that any packet do not have ip_opt already
79 +int pskb_allocate_ip_options(struct sk_buff *skb, u8 ip_opt_size,
80 +    gfp_t gfp_mask)
81 +{
82 + int i;
83 + u8 *data;
84 +
85 +#ifdef NET_SKBUFF_DATA_USES_OFFSET
86 + int size = ip_opt_size + skb->end;
87 +#else
88 + // i386 arch do not uses OFFSET
89 + int size = ip_opt_size + (skb->end - skb->head);
90 +#endif
91 + long off;
92 +
93 +
94 + // always keep transport_header correct (when forward a packet transport_header will not
       be setted
95 +#ifdef NET_SKBUFF_DATA_USES_OFFSET
96 + skb->transport_header = skb->network_header + 20;
97 +#else
98 + skb->transport_header = skb->head + (skb->network_header - skb->head) + 20;
99 +#endif
100 +
101 + BUG_ON(ip_opt_size < 0);
102 +
103 + if (skb_shared(skb))
104 +  BUG();
105 +
106 + size = SKB_DATA_ALIGN(size);
107 +
108 + // resides at skb->end
109 + data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
110 + if (!data)
111 +  goto nodata;
112 +
113 + /* Copy only real data... and, alas, header. This should be
114 + * optimized for the cases when header is void. */
115 + // skb->transport_header points to the start of bytes of transport_header, but...
116 + // in forward mode appears that it points to network_header
117 +#ifdef NET_SKBUFF_DATA_USES_OFFSET
118 + memcpy(data, skb->head, skb->transport_header);
119 + memcpy(data + ip_opt_size + skb->transport_header, skb->head +
       skb->transport_header, skb->tail - skb->transport_header);
120 +#else
121 + memcpy(data, skb->head, skb->transport_header - skb->head);
122 + memcpy(data + ip_opt_size + (skb->transport_header - skb->head), skb->head +
       (skb->transport_header - skb->head), skb->tail - skb->transport_header);
123 +#endif
```

```
124 + memcpy(data + size, skb_end_pointer(skb),
125 + sizeof(struct skb_shared_info));
126 +
127 + for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
128 +  get_page(skb_shinfo(skb)->frags[i].page);
129 +
130 + if (skb_has_frags(skb))
131 +  skb_clone_fraglist(skb);
132 +
133 + skb_release_data(skb);
134 +
135 + off = data - skb->head;
136 + skb->head = data;
137 + skb->data += off;
138 +#ifdef NET_SKBUFF_DATA_USES_OFFSET
139 + skb->end = size;
140 + off = ip_opt_size;
141 +#else
142 + skb->end = skb->head + size;
143 + /*
144 + why this is need? architectures < 33 bits do not uses offset, so
145 + we *need* to add offset when we realoc the skb.
146 + */
147 + skb->network_header += off;
148 + if (skb_mac_header_was_set(skb))
149 +  skb->mac_header += off;
150 + off += ip_opt_size;
151 +#endif
152 + /* {transport,network,mac}_header and tail are relative to skb->head */
153 + skb->transport_header += off;
154 + skb->tail += off;
155 +
156 + /* Only adjust this if it actually is csum_start rather than csum */
157 + if (skb->ip_summed == CHECKSUM_PARTIAL)
158 +  skb->csum_start += ip_opt_size;
159 +
160 + skb->cloned = 0;
161 + skb->hdr_len = 0;
162 + skb->nohdr = 0;
163 + atomic_set(&skb_shinfo(skb)->dataref, 1);
164 +
165 + return 0;
166 +
167 +nodata:
168 + return -ENOMEM;
169 +}
170 +EXPORT_SYMBOL(pskb_allocate_ip_options);
171 +
172  /**
173   * pskb_expand_head - reallocate header of &sk_buff
174   * @skb: buffer to reallocate
```

```
diff --git a/net/ipv4/ip_options.c b/net/ipv4/ip_options.c
index 4c09a31..b483602 100644
--- a/net/ipv4/ip_options.c
+++ b/net/ipv4/ip_options.c
@@ -42,12 +42,15 @@ void ip_options_build(struct sk_buff * skb, struct ip_options * opt,
        unsigned char *iph = skb_network_header(skb);

        memcpy(&(IPCB(skb)->opt), opt, sizeof(struct ip_options));
+
+ // iph have addr from start of network header
        memcpy(iph+sizeof(struct iphdr), opt->__data, opt->optlen);
+
        opt = &(IPCB(skb)->opt);

        if (opt->srr)
                memcpy(iph+opt->srr+iph[opt->srr+1]-4, &daddr, 4);

        if (!is_frag) {
                if (opt->rr_needaddr)
                        ip_rt_get_source(iph+opt->rr+iph[opt->rr+2]-5, rt);
@@ -119,6 +125,15 @@ int ip_options_echo(struct ip_options * dopt, struct sk_buff * skb)
                dptr += optlen;
                dopt->optlen += optlen;
        }
+ if (sopt->__pad2) {
+  //printk(KERN_EMERG "(ip_options_echo) echo for __pad2\n");
+  optlen = sptr[sopt->rr+1];
+  soffset = sptr[sopt->rr+2];
+  dopt->__pad2 = dopt->optlen + sizeof(struct iphdr);
+  memcpy(dptr, sptr+sopt->__pad2, optlen);
+  dptr += optlen;
+  dopt->optlen += optlen;
+ }
        if (sopt->ts) {
                optlen = sptr[sopt->ts+1];
                soffset = sptr[sopt->ts+2];
@@ -246,7 +261,10 @@ void ip_options_fragment(struct sk_buff * skb)
   * Caller should clear *opt, and set opt->data.
   * If opt == NULL, then skb->data should point to IP header.
   */
-
+// this really compile ipopt; it set pointers like ->rr and ->__pad2
+// that will be used to access options data. One thing: opt->__data
+// is used only to build an ip_option, so when you will access the
+// data you do not uses __data (talau)
  int ip_options_compile(struct net *net,
                         struct ip_options * opt, struct sk_buff * skb)
  {
@@ -444,3 +470,9 @@ int ip_options_compile(struct net *net,
                                goto error;
                        }
```

```
226                             break;
227 +  case IPOPT_NGWA:
228 +   //printk(KERN_EMERG "(ip_options_compile) IPOPT_NGWA\n");
229 +   opt->__pad2 = optptr - iph;
230 +   //sptr = skb_network_header(skb);
231 +   //printk(KERN_EMERG " opt->__pad2 %d;%d;%d\n", sptr[opt->__pad2+1],
        sptr[opt->__pad2+2], sptr[opt->__pad2+3]);
232 +   break;
233 @@ -462,6 +496,7 @@ eol:
234                         return 0;
235
236  error:
237 + printk(KERN_EMERG "(ip_options_compile) ERROR\n");
238           if (skb) {
239                   icmp_send(skb, ICMP_PARAMETERPROB, 0, htonl((pp_ptr-iph)<<24));
240           }
241 @@ -646,3 +682,7 @@ int ip_options_rcv_srr(struct sk_buff *skb)
242           }
243           return 0;
244  }
245 +
246 +// has NGWA module uses it, we *need* to export it
247 +// without this a error: "ERROR:" + "undefined!" + "make[1]: *** [__modpost] Error 1" +
        module build -Kconfig will show
248 +EXPORT_SYMBOL(ip_options_build);
249 diff --git a/net/ipv4/sysctl_net_ipv4.c b/net/ipv4/sysctl_net_ipv4.c
250 index 1cd5c15..575e609 100644
251 --- a/net/ipv4/sysctl_net_ipv4.c
252 +++ b/net/ipv4/sysctl_net_ipv4.c
253 @@ -484,6 +484,27 @@ static struct ctl_table ipv4_table[] = {
254                         .proc_handler = proc_dointvec,
255           },
256           {
257 +  .procname = "tcp_ngwa_on",
258 +  .data  = &sysctl_tcp_ngwa_on,
259 +  .maxlen  = sizeof(int),
260 +  .mode  = 0644,
261 +  .proc_handler = proc_dointvec,
262 + },
263 + {
264 +  .procname = "tcp_ngwa_mitigation",
265 +  .data  = &sysctl_tcp_ngwa_mitigation,
266 +  .maxlen  = sizeof(int),
267 +  .mode  = 0644,
268 +  .proc_handler = proc_dointvec,
269 + },
270 + {
271 +  .procname = "tcp_ngwa_mitigation_alpha",
272 +  .data  = &sysctl_tcp_ngwa_mitigation_alpha,
273 +  .maxlen  = sizeof(int),
274 +  .mode  = 0644,
```

```
275 +  .proc_handler = proc_dointvec,
276 + },
277 + {
278                         .procname = "tcp_mtu_probing",
279                         .data  = &sysctl_tcp_mtu_probing,
280                         .maxlen  = sizeof(int),
281 diff --git a/net/ipv4/tcp_input.c b/net/ipv4/tcp_input.c
282 index f240f57..861c54a 100644
283 --- a/net/ipv4/tcp_input.c
284 +++ b/net/ipv4/tcp_input.c
285 @@ -95,6 +95,8 @@ int sysctl_tcp_thin_dupack __read_mostly;
286  int sysctl_tcp_moderate_rcvbuf __read_mostly = 1;
287  int sysctl_tcp_abc __read_mostly;
288
289 +int sysctl_tcp_ngwa_on __read_mostly;
290 +
291  #define FLAG_DATA  0x01 /* Incoming frame contained data.  */
292  #define FLAG_WIN_UPDATE  0x02 /* Incoming ACK was a window update. */
293  #define FLAG_DATA_ACKED  0x04 /* This ACK acknowledged new data.  */
294 @@ -3781,6 +3783,8 @@ void tcp_parse_options(struct sk_buff *skb, struct
       tcp_options_received *opt_rx,
295                                         }
296                                         break;
297                                 case TCPOPT_WINDOW:
298 +   // today ngwa do not support scalled window
299 +   break;
300                                         if (opsize == TCPOLEN_WINDOW && th->syn &&
301                                             !estab && sysctl_tcp_window_scaling) {
302                                                 __u8 snd_wscale = *(__u8 *)ptr;
303 @@ -5221,10 +5225,50 @@ discard:
304    * the rest is checked inline. Fast processing is turned on in
305    * tcp_data_queue when everything is OK.
306    */
307 +// called every time in TCP connection (when it is established)
308  int tcp_rcv_established(struct sock *sk, struct sk_buff *skb,
309                                 struct tcphdr *th, unsigned len)
310  {
311          struct tcp_sock *tp = tcp_sk(sk);
312 +
313 + struct iphdr *iph;
314 + struct ip_options *opt;
315 + // will point to ip->opt real data
316 + unsigned char *sptr;
317 +
318 +// printk(KERN_EMERG "cwnd = %d\n", tp->snd_cwnd);
319 +
320 + iph = ip_hdr(skb);
321 + opt = &(IPCB(skb)->opt);
322 +
323 + tp->ngwa_ipopt = 0;
324 + if (sysctl_tcp_ngwa_on) {
```

```
+  printk(KERN_DEBUG "(tcp_rcv_established) NGWA is on\n");
+  if (opt && opt->optlen) {
+   // ngwa are in __pad2 pointer
+   if (opt->__pad2) {
+    sptr = skb_network_header(skb);
+    //printk(KERN_EMERG " (into) opt->__pad2 %d;%d;%d\n", sptr[opt->__pad2+1],
     sptr[opt->__pad2+2], sptr[opt->__pad2+3]);
+
+    if (sptr[opt->__pad2] == IPOPT_NGWA) {
+     //printk(KERN_EMERG "(tcp_rcv_established) NGWA found\n");
+     // 8 bits from __pad2+2 and 8 too from __pad2+3 = 16 bits
+     tp->ngwa_ipopt = (u16) ((u16) sptr[opt->__pad2+3] << 8) | (u16) sptr[opt->__pad2+2];
+    }
+   }
+
+   // for debug rr
+   /*
+   if (opt->rr) {
+   printk(KERN_EMERG "(tcp_rcv_established) have RR\n");
+   unsigned char *sptr;
+   sptr = skb_network_header(skb);
+   printk(KERN_EMERG "\t opt->rr %d;%d;%d\n", sptr[opt->rr+1], sptr[opt->rr+2]);
+   }
+   */
+  }
+ }
+
            int res;

            /*
diff --git a/net/ipv4/tcp_output.c b/net/ipv4/tcp_output.c
index 0dda86e..591887d 100644
--- a/net/ipv4/tcp_output.c
+++ b/net/ipv4/tcp_output.c
@@ -63,6 +63,11 @@ int sysctl_tcp_slow_start_after_idle __read_mostly = 1;
 int sysctl_tcp_cookie_size __read_mostly = 0; /* TCP_COOKIE_MAX */
 EXPORT_SYMBOL_GPL(sysctl_tcp_cookie_size);

+// in pt_BR mitigation = suavizacao
+int sysctl_tcp_ngwa_mitigation __read_mostly;
+// will be 0.3
+int sysctl_tcp_ngwa_mitigation_alpha __read_mostly = 3;
+

 /* Account for new data that has been sent to the network. */
 static void tcp_event_new_data_sent(struct sock *sk, struct sk_buff *skb)
@@ -212,6 +218,8 @@ void tcp_select_initial_window(int __space, __u32 mss,
                        (*rcv_wnd) = space;

            (*rcv_wscale) = 0;
+ // today ngwa do not support scalled window
```

```
+// wscale_ok = 0;
        if (wscale_ok) {
                /* Set window scaling on max possible window
                 * See RFC1323 for an explanation of the limit to 14
@@ -288,6 +296,32 @@ static u16 tcp_select_window(struct sock *sk)
        if (new_win == 0)
                tp->pred_flags = 0;

+ // today window scaling need to be off to ngwa work
+ if (sysctl_tcp_ngwa_on) {
+  // here we will implement sysctl_tcp_ngwa_mitigation
+  printk(KERN_DEBUG "(tcp_select_window) NGWA is on\n");
+  if (tp->ngwa_ipopt) {
+   printk(KERN_DEBUG "(tcp_select_window) cur_win %d; ngwa_ipopt = %d\n", new_win,
      tp->ngwa_ipopt);
+
+   if (sysctl_tcp_ngwa_mitigation) {
+    printk(KERN_DEBUG "(tcp_select_window) ngwa-mitigation is on\n");
+    printk(KERN_DEBUG "\t alpha %d; ngwa_avg %d; ngwa_ipopt %d",
     sysctl_tcp_ngwa_mitigation_alpha, tp->ngwa_avg, tp->ngwa_ipopt);
+
+    // Eqn Modified from original; but is the same
+    // (alpha=0.1 .. 0.9) orig: Wngwa1 = (1 - alpha) * Wngwa1 + alpha * Wngwa
+    // (alpha=1 .. 9) new: Wngwa1 = (10 - alpha) * 0.1 * Wngwa1 + (alpha * 0.1) * Wngwa
+    tp->ngwa_avg = (u32) ((10 - sysctl_tcp_ngwa_mitigation_alpha) * tp->ngwa_avg +
     sysctl_tcp_ngwa_mitigation_alpha * tp->ngwa_ipopt) / 10;
+    printk(KERN_DEBUG "(tcp_select_window) ngwa_avg = %d\n", tp->ngwa_avg);
+    new_win = min_t(u32, new_win, tp->ngwa_avg);
+    printk(KERN_DEBUG "(tcp_select_window) new_win = %d\n", new_win);
+   } else {
+    new_win = min_t(u32, new_win, tp->ngwa_ipopt);
+    printk(KERN_DEBUG "(tcp_select_window) new_win = %d\n", new_win);
+   }
+  }
+ }
+
+ ////printk(KERN_DEBUG "(tcp_select_window) tcp->window = %d\n", new_win);
        return new_win;
 }

@@ -525,12 +559,14 @@ static void tcp_options_write(__be32 *ptr, struct tcp_sock *tp,
                                        TCPOLEN_SACK_PERM);
        }

+ /*
        if (unlikely(OPTION_WSCALE & options)) {
                *ptr++ = htonl((TCPOPT_NOP << 24) |
                                (TCPOPT_WINDOW << 16) |
                                (TCPOLEN_WINDOW << 8) |
                                opts->ws);
        }
```

```
+ */

        if (unlikely(opts->num_sack_blocks)) {
                struct tcp_sack_block *sp = tp->rx_opt.dsack ?
@@ -586,6 +622,8 @@ static unsigned tcp_syn_options(struct sock *sk, struct sk_buff
    *skb,
             * SACKs don't matter, we never delay an ACK when we have any of those
             * going out. */
            opts->mss = tcp_advertise_mss(sk);
+ opts->mss -= NGWA_OLEN;
+//// printk(KERN_EMERG "SYN tcp mms = %d (%u)\n", opts->mss, opts->mss);
            remaining -= TCPOLEN_MSS_ALIGNED;

            if (likely(sysctl_tcp_timestamps && *md5 == NULL)) {
@@ -688,6 +726,8 @@ static unsigned tcp_synack_options(struct sock *sk,

            /* We always send an MSS option. */
            opts->mss = mss;
+ opts->mss -= NGWA_OLEN;
+//// printk(KERN_EMERG "SYN/ACK tcp mms = %d (%u)\n", opts->mss, opts->mss);
            remaining -= TCPOLEN_MSS_ALIGNED;

            if (likely(ireq->wscale_ok)) {
diff --git a/net/sched/Kconfig b/net/sched/Kconfig
index 2f691fb..2bac93e 100644
--- a/net/sched/Kconfig
+++ b/net/sched/Kconfig
@@ -126,6 +126,16 @@ config NET_SCH_RED
              To compile this code as a module, choose M here: the
              module will be called sch_red.

+config NET_SCH_NGWA
+ tristate "New Generalized Window Advertising (NGWA)"
+ ---help---
+ NGWA
+
+ See the top of <file:net/sched/sch_ngwa.c> for more details.
+
+ To compile this code as a module, choose M here: the
+ module will be called sch_ngwa.
+
 config NET_SCH_SFQ
            tristate "Stochastic Fairness Queueing (SFQ)"
            ---help---
diff --git a/net/sched/Makefile b/net/sched/Makefile
index f14e71b..c72b283 100644
--- a/net/sched/Makefile
+++ b/net/sched/Makefile
@@ -20,6 +20,7 @@ obj-$(CONFIG_NET_SCH_CBQ) += sch_cbq.o
 obj-$(CONFIG_NET_SCH_HTB) += sch_htb.o
 obj-$(CONFIG_NET_SCH_HFSC) += sch_hfsc.o
```

```
473   obj−$(CONFIG_NET_SCH_RED) += sch_red.o
474  +obj−$(CONFIG_NET_SCH_NGWA) += sch_ngwa.o
475   obj−$(CONFIG_NET_SCH_GRED) += sch_gred.o
476   obj−$(CONFIG_NET_SCH_INGRESS) += sch_ingress.o
477   obj−$(CONFIG_NET_SCH_DSMARK) += sch_dsmark.o
```