Avaliação por Simulação dos Mecanismos de Controle de Congestionamento do TCP

UFRJ

Projeto submetido para a obtenção do título de

Engenheiro Eletrônico

ao Departamento de Eletrônica

da Escola de Engenharia da UFRJ

por

Guilherme Paulo Teixeira dos Santos e

Saulo Vaz de Vasconcellos

Avaliação por Simulação dos Mecanismos de Controle de Congestionamento do TCP

Autores:

Guilherme Paulo Teixeira dos Santos

Saulo Vaz de Vasconcellos

Orientador:

Prof. José Ferreira de Rezende, Dr.

Examinadores:

Prof. Aloysio de Castro Pinto Pedroza, Dr.

Eng. Artur Ziviani, M. Sc.

Departamento de Eletrônica - Escola de Engenharia - UFRJ

Rio de Janeiro, RJ - Brasil

Agosto de 2000

Aos nossos pais.

Agradecimentos

Aos nossos pais, Joselina e Luiz Diogo, Terezinha e Paulo Sergio, e a todos os familiares, pelo apoio e incentivo ao longo de toda a vida.

Ao professor José Ferreira de Rezende, nosso orientador.

Aos membros da banca examinadora, Aloysio e Artur, pela dedicação na leitura e pelas críticas construtivas que contribuiram para o resultado final do trabalho.

Aos colegas formandos do segundo período de 1998, especialmente ao Rafael "Rosinha" e Zé Carlos, companheiros de estudo e de inúmeros trabalhos de grupo durante o curso. Aos amigos e namoradas, pela paciência e apoio ao longo de todo o curso de engenharia eletrônica.

Aos professores do Departamento de Eletrônica da Escola de Engenharia da UFRJ, especialmente aos professores Aloysio, Otto, Seixas, Mauros, Gelson, Casé, Rhomberg e Eduardo. Agradecemos também à Dona Conceição pela presteza no atendimento na secretaria do Departamento.

Aos colegas, membros do Grupo de Teleinformática e Automação (GTA) da COPPE/PEE, que incentivaram o trabalho e que estiveram sempre prontos a ajudar e a esclarecer quaisquer dúvidas que surgissem, das mais simples às mais complexas. São eles Marcial, Alexandre S., Rubi, Marcos Tadeu, Vidal, Belém, Alexandre R., Daniel, Ezequiel, Davi, Fabrício, Jaime, Viviane, Bagatelli, Bernardo, Doc, Renzo, Ingrid, Juliana, e em especial, Gardel, Roman, Valentim, Paulo, Artur, Marcio, Eric, Pedro, Aline, Kleber, Fagundes e Granato. Aos colegas da COPPE/Sistemas, Igor e Robson.

Resumo do Projeto Final apresentado ao DEL/UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletrônico.

Avaliação por Simulação dos Mecanismos de Controle de Congestionamento do TCP

Guilherme Paulo Teixeira dos Santos e

Saulo Vaz de Vasconcellos

Agosto de 2000

Orientador: José Ferreira de Rezende

Departamento: Engenharia Eletrônica

As redes de computadores já são parte integrante do nosso cotidiano. A Internet, interligando os mais diversos tipos de sistemas e oferecendo os mais diversos serviços, vem crescendo cada vez mais. Essa popularização da Internet acarreta o aumento de demanda na utilização da rede, exigindo eficiência de sua infra-estrutura. Os serviços oferecidos pelo protocolo TCP representam 95% do tráfego da rede mundial. Desta forma a eficiência da Internet está intimamente ligada à eficiência desse protocolo. Os mecanismos de controle de congestionamento do TCP são então o foco principal deste trabalho. Uma visão geral do protocolo é apresentada a fim de se criar o contexto. Além da introdução de alguns conceitos envolvidos em controle de congestionamento, são explicados os mecanismos utilizados para resolver o problema em diferentes implementações do TCP atualmente em uso. Simulações de cenários típicos de congestionamento são efetuadas e analisadas de forma minuciosa, permitindo a compreensão dos detalhes do funcionamento dos algoritmos. Nas conclusões deste trabalho apresenta-se uma análise comparativa entre as implementações do TCP atualmente existentes.

Palavras Chave

Redes de computadores

Internet

TCP

Controle de congestionamento

Simulação

Sumário

Lista de Figuras

LISTA DE FIGURAS

Lista de Tabelas

Capítulo I

Introdução

Nos dias atuais, a informática encontra-se presente em quase todos os campos da atividade humana. Esta atuação das ferramentas da computação em diferentes áreas criou a necessidade da comunicação entre os diversos sistemas, dando origem às redes de computadores. A Internet aparece neste contexto como a rede mundial de maior abrangência e importância.

Dentre as utilizações da Internet, as aplicações de WWW (*World Wide Web*) são as mais representativas. Tais aplicações utilizam para seu funcionamento os serviços oferecidos pelo protocolo TCP (*Transmission Control Protocol*), como por exemplo o transporte confiável dos dados. Atualmente, em um enlace típico da Internet, medidas indicam que 95% dos bytes transportados são de tráfego TCP, sendo que destes, 75% são provenientes de aplicações HTTP [1]. Assim fica claro que o desempenho do protocolo TCP é de fundamental importância para o desempenho global da Internet.

O grande crescimento que a Internet vem sofrendo aumenta a carga de utilização da rede. Seu crescimento vem ocorrendo de forma exponencial, onde o número de *hosts* dobrou a cada 18 meses nos últimos três anos. Desta forma, em janeiro de 2000 a Internet já alcançava mais de 72 milhões de *hosts* [2]. Este fato causa situações de congestionamento que acabam por reduzir em muito a eficiência da utilização da rede. Desta forma é de extrema importância que mecanismos sejam incluídos/alterados para que se possa obter um certo controle destas situações.

Desde a sua especificação original, no início dos anos 80, o protocolo TCP tem sofrido diversas modificações, gerando melhorias que permitem a sua utilização em ambientes cada vez mais heterogêneos. Dentre as diversas modificações sofridas por este protocolo, a de maior importância foi o acréscimo de mecanismos de controle de congestionamento realizada em 1988 por Van Jacobson [3]. Nesta época a Internet vinha sofrendo de graves problemas de desempenho, ocorridos devido à ausência de tais mecanismos. Os usuários competiam de maneira extremamente agressiva pelos recursos da rede, gerando instabilidade e colapsos de congestionamento. As modificações sugeridas por [3] representaram uma melhoria da ordem de 10 vezes no desempenho do TCP. Desde então, vários pesquisadores têm trabalhado na concepção de novos algoritmos e heurísticas, objetivando um maior desempenho, adaptados às novas condições de tráfego na Internet. Infelizmente, a maioria destas propostas não puderam ser implantadas em âmbito global, dada a importância adquirida pela Internet nos últimos anos, onde qualquer modificação poderia significar um rompimento no serviço fornecido.

Avanços recentes nas tecnologias de comutação e de transmissão permitem a implementação de redes de alta velocidade que estão mudando a face da rede mundial Internet. Estas novas tecnologias têm permitido um aumento constante da capacidade das espinhas dorsais desta infra-estrutura. E agora, com o surgimento da Internet 2, de vocação explicitamente acadêmica e experimental, a pesquisa envolvendo a Internet adquire uma nova força.

I.1 Objetivo

Este trabalho tem como objetivo avaliar o funcionamento e o desempenho de uma gama de novos mecanismos propostos para a melhoria de desempenho do protocolo TCP. Estes mecanismos, na sua maioria, modificam o esquema de controle de congestionamento existente.

Desta forma, esse trabalho realiza um estudo das diferentes implementações do TCP. Esse protocolo é o responsável pela detecção de erros, ordenação de pacotes, controle de fluxo e controle de congestionamento. Os diferentes mecanismos de controle de congestionamento do TCP são o principal foco deste trabalho. Estes algoritmos têm como objetivo otimizar a vazão, mantendo alta a eficiência de utilização do canal e ao mesmo tempo evitando que os diversos fluxos concorrentes causem congestionamento na rede. Pelas características altamente dinâmicas do tráfego de dados em uma rede, faz-se necessário o uso de algoritmos adaptativos para a realização destes objetivos.

Dentre os mecanismos propostos como modificação do protocolo original, podemos destacar os implementados nas seguintes versões do TCP: Tahoe [4], Reno [5], New Reno [6], Vegas [7] e TCP SACK [8], que foram escolhidos para serem abordados neste trabalho.

Para a avaliação das implementações citadas foram realizadas simulações de cenários típicos de congestionamento em redes de pacotes. Para tanto foi utilizado o simulador de redes *NS* (*network simulator*) [9], desenvolvido em Berkeley.

I.2 Estrutura do Texto

O foco principal do trabalho são os mecanismos de controle de congestionamento do TCP, como já foi colocado anteriormente. Não seria adequado, entretanto, que se abordasse diretamente um item tão específico do protocolo sem que se tivesse uma idéia geral de seu funcionamento. Desta forma, no capítulo II, é abordado o funcionamento do protocolo TCP de uma maneira geral. Este capítulo não tem a pretensão de explicar detalhadamente o funcionamento do TCP, mas sim de dar uma noção de como o protocolo se comporta, permitindo que se tenha uma visão mais completa e que se compreenda o controle de congestionamento inserido no contexto do funcionamento do TCP.

Já o capítulo III aborda diretamente o problema do congestionamento e os algoritmos desenvolvidos para sua solução. Primeiramente mecanismos sugeridos por [3] são abordados de forma individual, tendo seu funcionamento detalhadamente explicado. Em seguida, as combinações dos diversos algoritmos são apresentadas, constituindo as versões existentes do protocolo. Por fim alguns esquemas de controle de congestionamento posteriores à [3] são colocados de forma detalhada.

As simulações realizadas são mostradas no capítulo IV. Inicialmente, o simulador de redes é brevemente apresentado, para ser posteriormente detalhado, no Apêndice A. A topologia proposta para o estudo é então justificada e as simulações para diferentes padrões de perdas são colocadas. Neste momento, é realizada uma análise do funcionamento dos diversos algoritmos de controle de congestionamento que busca esclarecer as diferenças qualitativas de cada implementação do protocolo. Em seguida é apresentada uma outra simulação que visa analisar o desempenho de diferentes implementações do TCP na transmissão de um arquivo. No final deste capítulo os comportamentos das implementações do protocolo são então resumidos, com base nas observações das simulações.

Por fim, o capítulo V apresenta as conclusões do trabalho, comparando as diferentes implementações do TCP, permitindo uma análise das vantagens e problemas de cada uma delas. Desta forma procurou-se escrever um texto estruturado, iniciando com uma idéia geral sobre o protocolo TCP, chegando aos mecanismos de controle de congestionamento e acompanhando o funcionamento destes com a análise detalhada das simulações.

Capítulo II

Protocolo TCP

Neste capítulo o protocolo TCP é apresentado de forma breve, buscando-se dar uma idéia geral, mostrando-se os princípios básicos do seu funcionamento. Inicialmente é dada uma descrição geral do TCP. Em seguida, explica-se como esse protocolo provê confiabilidade em suas conexões. O mecanismo de janelas deslizantes, para melhorar o desempenho nas transmissões, é abordado a seguir. Os campos do segmento TCP são então detalhados e por fim os procedimentos de estabelecimento e término de conexões TCP são brevemente explicados.

II.1 Descrição Geral

A pilha de protocolos da arquitetura TCP/IP contém dois protocolos na camada de transporte (figura II.1): UDP (*User Datagram Protocol*) e TCP (*Transmission Control Protocol*). Apesar das inúmeras diferenças entre os dois protocolos, pode-se afirmar que a maior delas consiste na garantia ou não da confiabilidade no recebimento dos dados por parte do ponto destino.

Os protocolos da camada transporte, o UDP e o TCP, se utilizam dos serviços da

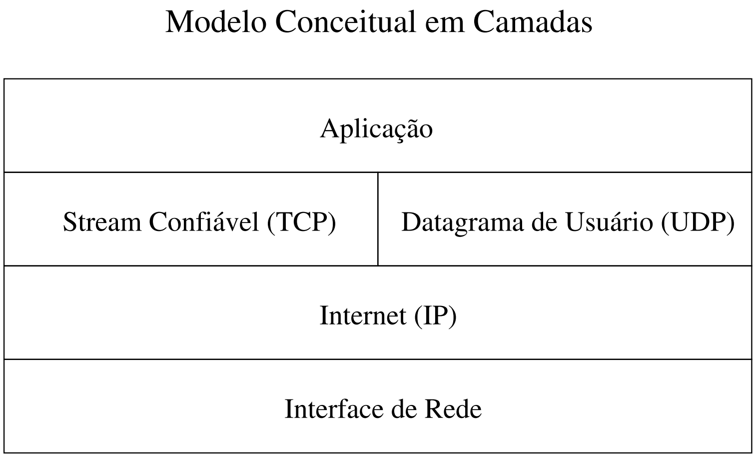


Figura II.1: Camadas da arquitetura TCP/IP.

camada inferior, no caso, os serviços de uma rede IP. O protocolo IP que por sua vez se baseia no princípio de melhor esforço (*best-effort*), fornece um serviço não confiável, onde pacotes são descartados em caso de esgotamento de recursos. O protocolo UDP não faz uso de nenhum mecanismo para prover confiabilidade na transmissão dos dados. Já o protocolo TCP, através de um esquema de reconhecimentos positivos e retransmissões, garante que todos os segmentos[1] sejam entregues ao nó destino.

Além de prover um serviço confiável podemos citar outras características importantes do protocolo TCP:

- Orientado a conexão: antes das duas aplicações iniciarem a transmissão, elas devem estabelecer uma conexão, como ocorre em telefonia por exemplo. Durante o estabelecimento da conexão, alguns parâmetros referentes à transmissão dos dados são determinados. A conexão é terminada quando umas das partes decide fazê-lo. É importante ressaltar que as conexões são *full-duplex*, sendo as transmissões de dados nos dois sentidos totalmente independentes. No entanto, o protocolo permite que dados de controle da transmissão em um determinado sentido possam ser enviados através de dados que trafegam no sentido oposto (*piggybacking*). Vale ressaltar que

[1] Segmento é o nome dado à Unidade de Dados do Protocolo ou PDU (*Protocol Data Unit*) da camada de transporte.

o uso de *piggybacking* não é obrigatório porque é possível que não ocorra transmissão de dados em um dos sentidos.

- Orientado a sequência (*streams*) de bytes não estruturados: os dados são enviados como uma seqüência de bytes, não contendo nenhum tipo de estrutura adicional. O destino se encarrega somente em entregar os bytes para a aplicação na mesma ordem em que eles foram transmitidos.

- Utilização de *buffers*: os dados da aplicação transmissora são enviados para o software TCP em quantidades de bytes escolhidas da forma mais conveniente, variando de aplicação para aplicação. O software TCP, por sua vez, ao receber os dados da aplicação transmissora, os acumula em um *buffer* de transmissão. Eles são então enviados em blocos cujo tamanho geralmente difere daquele com que ele recebe os dados da aplicação, objetivando a otimização da taxa de transferência dos dados. O mesmo raciocínio vale para o nó destino, ou seja, os dados são armazenados em um *buffer* de recepção e lidos pela aplicação, não necessariamente em blocos com o mesmo tamanho que os utilizados na transmissão através da rede.

Além das principais características, faz-se necessário esclarecer exatamente em que consiste o protocolo TCP, independentemente de suas diferentes implementações. O TCP é um protocolo de comunicação e como tal consiste em uma série de especificações que devem ser seguidas. Porém, a forma como estas especificações serão implementadas pode variar. Sendo assim, não se deve confundir o protocolo TCP com um trecho de código de uma de suas implementações.

O protocolo TCP define o formato dos dados e reconhecimentos (informação de controle) que as aplicações utilizam para atingir um serviço confiável, assim como os procedimentos efetuados para assegurar o recebimento correto dos dados. O protocolo especifica também como a entidade TCP distingue entre múltiplos destinos em um mesmo nó, como ele trata erros (duplicação, por exemplo) e perdas de pacotes e como inicia e termina uma conexão.

Apesar de especificar as operações efetuadas, o protocolo não especifica como a aplicação deve acessar estas operações e conseqüentemente como ter acesso aos serviços oferecidos pelo protocolo TCP. Sendo assim, há liberdade para implementação das interfaces das aplicações com o software TCP.

O protocolo TCP é bastante robusto, podendo ser usado em várias tecnologias de redes diferentes, no que se refere a topologias, velocidades e retardos de transmissão. Sua importância se reflete na presença no nome da própria arquitetura TCP/IP, que é a base da Internet como conhecida nos dias atuais, onde mais de 90% do tráfego é constituido por pacotes TCP [1].

As próximas seções irão abordar com mais detalhes características do protocolo TCP.

II.2 Provendo Confiabilidade

Como colocado anteriormente, o TCP se utiliza dos serviços do protocolo da camada de rede, o protocolo IP, que não oferece nenhum tipo de garantia para a entrega dos pacotes. Assim o protocolo TCP necessita de mecanismos próprios para a garantia de confiabilidade. Para atingir este objetivo, o protocolo TCP executa as seguintes tarefas:

- Divide os dados passados pela aplicação em blocos de tamanho conveniente. Estes blocos são denominados segmentos (ver seção II.4).

- Quando o TCP recebe dados do outro ponto-final da conexão, ele envia um reconhecimento. Os reconhecimentos podem ser enviados imediatamente ou após um atraso devidamente escolhido. Este último procedimento é conhecido como mecanismo de reconhecimentos atrasados (*delayed acks*) e visa aumentar a eficência da conexão, utilizando apenas um pacote para reconhecer uma quantidade maior de dados. É importante se observar que este último reconhecimento recebido é um reconhecimento cumulativo, ou seja, ele reconhece a chegada de todos os segmentos até aquele que desencadeou a geração deste sinal de reconhecimento.

- A cada segmento enviado, um temporizador associado é disparado. O TCP então espera pela confirmação do recebimento deste segmento pelo nó destino. Caso o tempo se esgote sem que o reconhecimento tenha chegado, o segmento é retransmitido.

- Mantém um mecanismo de proteção da integridade para o cabeçalho, através de um mecanismo de *checksum*, de forma a detectar qualquer alteração no conteúdo da informação transmitida. Caso haja algum erro, o pacote é descartado no destino e a sua retransmissão é aguardada.

- Por utilizarem os serviços de uma rede IP, os segmentos podem chegar ao destino fora de ordem. O TCP deve efetuar então a reordenação dos dados a fim de garantir a entrega destes em ordem para a aplicação. A infra estrutura de rede pode ainda causar a duplicação de segmentos [5]. Este fato não é tratado pelo protocolo IP, fazendo com que o TCP tenha que descartar dados duplicados, de forma transparente à aplicação.

- Exerce um controle de fluxo fim-a-fim fazendo com que o nó destino permita que o nó fonte envie no máximo a quantidade de bytes correspondente ao espaço livre no *buffer* de recepção.

Dentre as tarefas citadas acima, no que se refere à confiabilidade, pode-se dizer que as mais importantes são a segunda e terceira, as quais em conjunto formam o conceito de reconhecimento positivo e retransmissão. Positivo aqui se refere ao fato do receptor enviar reconhecimentos correspondentes aos segmentos recebidos, em oposição ao caso onde o destino implementaria os temporizadores e enviaria reconhecimentos negativos com relação a segmentos que não tivessem sido recebidos até o esgotamento do temporizador.

A figura II.2 mostra como este esquema garante que todos os segmentos serão recebidos em algum momento, inclusive em casos onde ocorre o estouro do temporizador. Pode-se notar que nesta figura os segmentos e os reconhecimentos são numerados. Isto é necessário para que o nó fonte possa distinguir para qual segmento se refere cada reconhecimento. Assim os pacotes de reconhecimento contém um campo indicando o segmento recebido com sucesso que foi responsável por sua geração. Este exemplo da figura II.2 é o caso de transmissão *stop-and wait* onde o emissor aguarda o recebimento de um sinal de reconhecimento para só então enviar um novo pacote de dados.

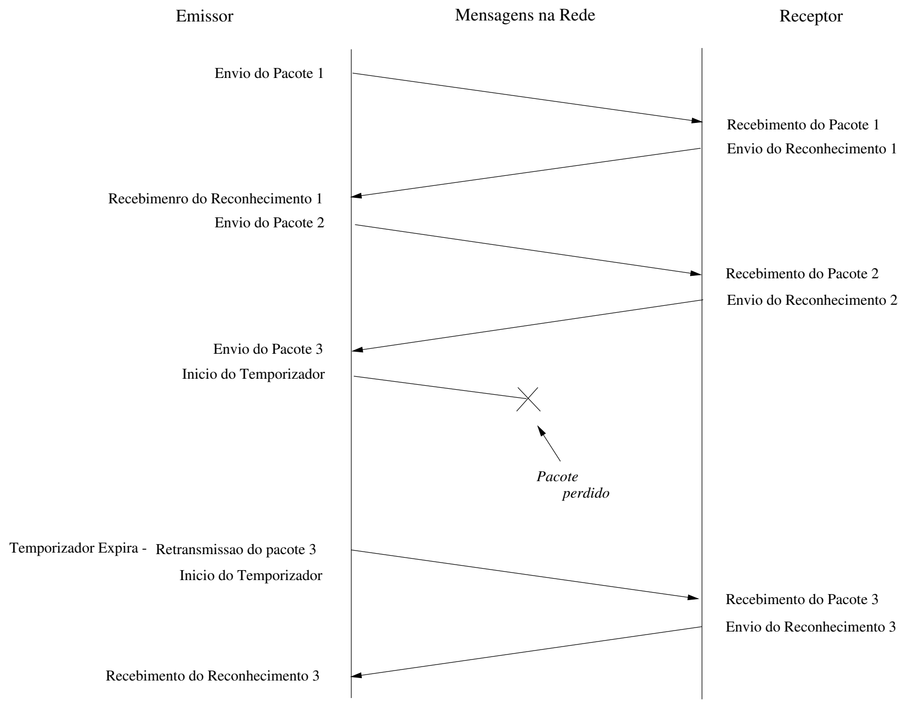


Figura II.2: Reconhecimento positivo e retransmissão.

II.3 Janelas Deslizantes

A política de reconhecimentos *stop-and-wait* descrita na seção anterior, onde o emissor aguarda o recebimento do reconhecimento do pacote enviado para realizar o envio de um novo segmento não é a melhor estratégia para a transmissão confiável de dados. Na realidade ocorre uma grande ineficiência quanto à utilização do canal. Isto se dá porque a rede fica ociosa, desperdiçando banda passante enquanto aguarda a chegada de um reconhecimento, para só então enviar novos dados. Para contornar essa limitação, o protocolo TCP utiliza um mecanismo de envio de pacotes em avanço, através da utilização de janelas deslizantes para a transmissão.

A utilização desta técnica permite um melhor aproveitamento da capacidade do canal, uma vez que são enviados pacotes mesmo durante a espera por reconhecimentos positivos de dados enviados anteriormente, o que mantém o meio ocupado de modo mais eficiente. Para ilustrar este fato podemos comparar a figura II.2, que se refere ao esquema explicado anteriormente, com a figura II.3, representando um protocolo que permita o envio de segmentos em avanço. Podemos observar claramente que a taxa de ocupação no meio é muito maior no segundo caso.

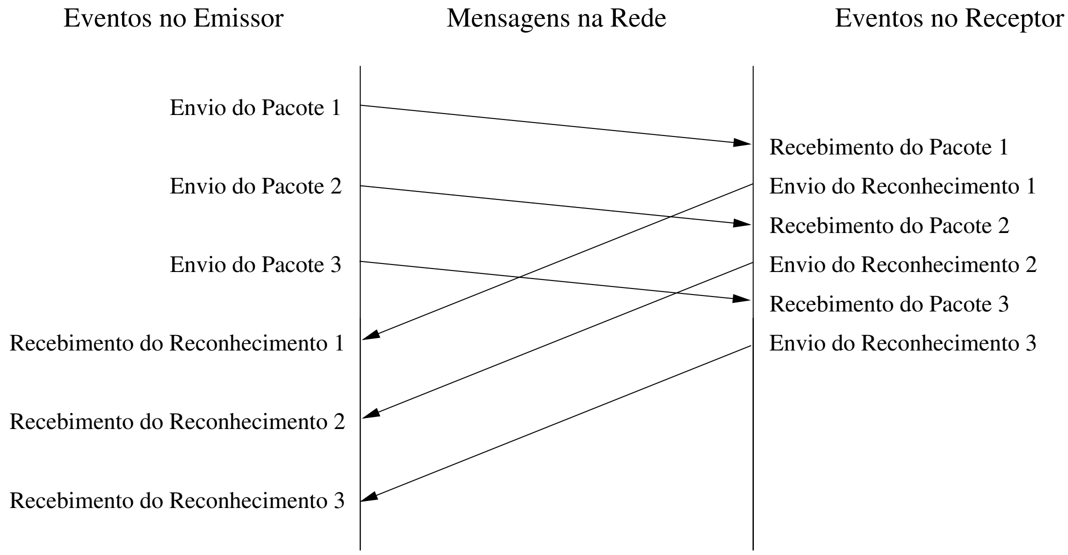


Figura II.3: Aumento da vazão com o envio de segmentos em avanço.

Para a compreensão do mecanismo de envio de pacotes em avanço, utilizando janelas deslizantes, podemos modelar os dados a serem transmitidos como uma fila de pacotes. Uma janela de dados, com uma certa quantidade de pacotes, poderá ser então enviada num dado instante. A figura II.4 ilustra a situação.

Ao receber os reconhecimentos dos segmentos enviados, a janela desloca-se permitindo o envio de novos dados, que estavam à sua direita. No caso da figura II.4 poderíamos enviar o pacote de número 9 ao receber o reconhecimento do pacote de número 1, deslizando a janela de transmissão. Este esquema faz com que a borda esquerda da janela indique o pacote que foi enviado há mais tempo e que ainda não foi reconhecido pelo receptor. Já a borda direita da janela indica o segmento de maior número que pode ser enviado num dado momento, considerando-se os reconhecimentos recebidos até aquele instante.

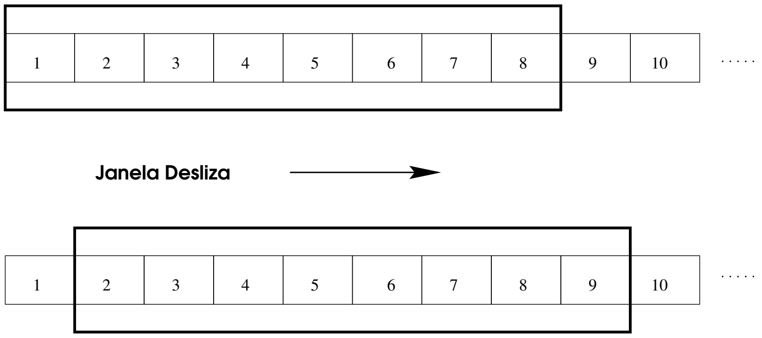


Figura II.4: Esquema do funcionamento da janela deslizante.

Podemos notar que sem a utilização de janelas deslizantes, quando um novo pacote só é enviado após o reconhecimento do anterior, a vazão máxima obtida seria a indicada na equação II.1, onde MSS representa o tamanho máximo do segmento em bytes.

$$Vaz\tilde{a}o_{max} = \frac{MSS * 8}{RTT} bits/seg \tag{II.1}$$

Já com a utilização de janelas deslizantes, nota-se que o limite teórico para a vazão pode ser aumentado, ficando dependente do tamanho da janela de transmissão como pode ser visto na equação II.2.

$$Vaz\tilde{a}o_{max} = \frac{windowsize * 8}{RTT} bits/seg \tag{II.2}$$

Notemos que *window size* é o tamanho máximo da janela de transmissão dado em Bytes. Este valor é determinado de forma dinâmica, sendo fornecido pelo receptor, indicando o tamanho disponível do seu *buffer*. No TCP, a variável *awnd* (de *advertised window*, a janela anunciada pelo receptor) é que representa este valor.

A janela anunciada pelo receptor (*awnd*), visa realizar um controle de fluxo no TCP. Como o protocolo pode ser utilizado em ambientes heterogêneos, onde os pontos-finais da conexão podem possuir recursos como *buffers* e capacidade de processamento diferentes, podem ocorrer casos onde o emissor envie os segmentos fazendo com que estes cheguem ao receptor numa taxa superior à capacidade de processamento deste último. Neste caso é necessário que o receptor disponha de meios de conter a taxa de envio do emissor, realizando um ajuste de transmissão de modo à evitar o esgotamento de seus *buffers*. Assim, em todos os sinais de reconhecimento, o receptor informa ao transmissor a quantidade de memória livre nos *buffers* de recepção. Desta forma a janela de transmissão fica limitada, no máximo, à quantidade de bytes disponíveis nos *buffers* do receptor.

Para que ocorra uma ocupação eficiente do canal de transmissão, o valor da janela de transmissão deve ser ajustado cuidadosamente. Deve-se evitar uma possível sobrecarga da rede, com o uso de janelas demasiadamente grandes, como também a sub-utilização do meio, através do emprego de janelas pequenas.

II.4 O Segmento TCP

Esta seção descreve os campos do segmento TCP, utilizados para iniciar conexões, transferir dados, enviar reconhecimentos, anunciar tamanhos de janelas e terminar conexões.

Cada segmento TCP é dividido em duas partes: cabeçalho e dados (figura II.5). Os dois primeiros campos identificam os números das portas nos nós fonte e destino, respectivamente. Para compreender estes campos, é necessário que se entenda como o protocolo TCP define e utiliza os conceitos de porta e conexão.

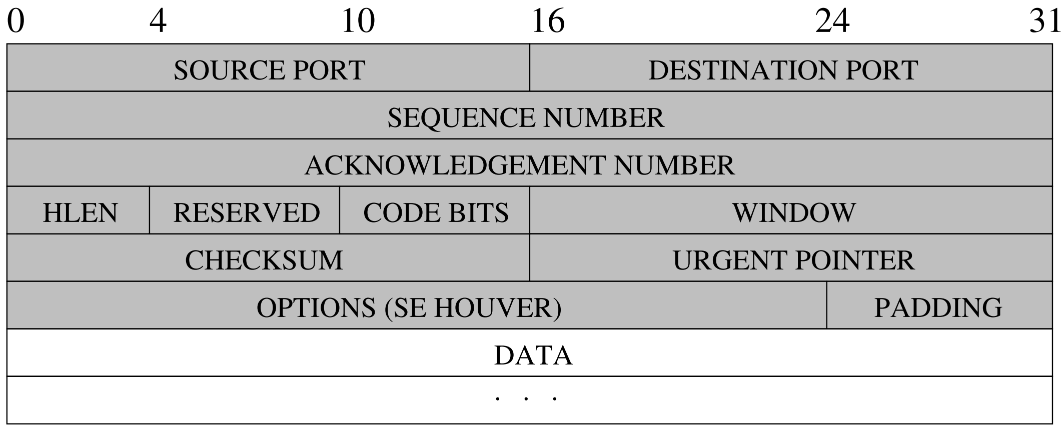


Figura II.5: Formato do segmento TCP.

Para atender ao fato de que diferentes aplicações em uma determinada estação podem estar se comunicando com outras estações na rede num dado momento, o TCP se utiliza do endereçamento por portas. A fim de identificar para qual aplicação deve ser entregue um determinado segmento, o protocolo TCP utiliza o número de porta ao lado do endereço IP da estação, definindo assim um ponto-final (*endpoint*) como o par (`estação, porta`).

Esta não é ainda a abstração fundamental do TCP. O protocolo identifica uma conexão através de um par de pontos-finais. Isto permite que uma mesma porta seja compartilhada por diferentes conexões que desejam acessar uma mesma aplicação simultaneamente, sem que ocorram ambiguidades. Isto ocorre, por exemplo, no caso de uma aplicação servidora de FTP. Um dos pontos-finais é sempre o par (endereço IP da estação servidora, porta padrão). O outro ponto-final é que evita a ambiguidade na identificação da conexão. A tabela II.1 ilustra com exemplos estes casos, onde diferentes conexões utilizam um ponto-final em comum (no caso o "ponto-final destino").

Tabela II.1: Exemplos de conexões TCP.

PONTO FINAL FONTE	PONTO FINAL DESTINO
$(10.10.0.10, 1048)$	$(157.148.10.57, 53)$
$(10.10.0.10, 1094)$	$(157.148.10.57, 53)$
$(10.10.0.50, 1048)$	$(157.148.10.57, 53)$

Alguns números de porta são conhecidos de antemão para as aplicações mais freqüentemente utilizadas, tais como correio eletrônico, DNS, telnet, FTP, etc.

Desta forma, como explicado acima, o protocolo TCP utiliza os campos SOURCE PORT e DESTINATION PORT (presentes no segmento TCP) e os endereços IP (presentes no cabeçalho do pacote IP) para identificar a qual conexão pertencem os dados. Nota-se aqui uma violação das camadas, pois a informação da camada rede (endereço IP) é utilizada na camada transporte.

Prosseguindo pelo cabeçalho do segmento TCP, o campo SEQUENCE NUMBER (número de seqüência) identifica a posição, dentro do *stream* de dados, relativa ao primeiro byte que está sendo enviado ao outro lado da conexão. Já o campo ACKNOWLEDGEMENT NUMBER (número de reconhecimento) identifica o número do próximo byte que o nó destino espera receber.

Para ser mais exato, o esquema de reconhecimento do protocolo TCP é cumulativo porque ele reconhece o último byte recebido de forma contínua. Sendo assim, se algum segmento for perdido, pode ser que haja lacunas no *buffer* de recepção, fazendo com que os dados com os bytes de número seqüência maiores do que os referentes à lacuna não sejam reconhecidos, apesar de poderem ser armazenados.

Este esquema apresenta vantagens e desvantagens. Dentre as vantagens estão a facilidade de geração (um único número) e o fato de nem sempre a perda de um reconhecimento provocar retransmissão, pois se um próximo segmento chegar antes que algum temporizador se esgote, ele reconhecerá também todos os dados que seriam recebidos anteriormente.

A grande desvantagem está no fato do nó fonte não ter o controle de todas as transmissões realizadas com sucesso. Isto pode acarretar retransmissões desnecessárias ou espera de próximos reconhecimentos para saber se o nó destino conseguiu receber e armazenar os dados transmitidos. Ambos os esquemas são ineficientes. Isto é um dos motivos que reforçam a idéia de introduzir reconhecimentos seletivos no protocolo TCP (SACK - *Selective Acknowledgement*), conforme abordado na seção III.3.

O campo HLEN (*header length*) contém 4 bits e indica o tamanho do cabeçalho em blocos múltiplos de 32 bits. Isto é preciso porque o tamanho do campo OPTIONS (opções) pode variar dependendo das opções do protocolo TCP. O campo RESERVED contém 6 bits e, conforme diz o nome, está reservado para uso futuro. O campo CODE BITS contém os bits de código para identificar o propósito dos dados que estão sendo enviados através dos segmentos. A tabela II.2 resume o significado destes bits cuja utilização será detalhada a seguir.

O bit URG (urgente) faz com que seja possível para o sofware TCP no nó destino sinalizar para a aplicação sobre a presença de dados prioritários. A aplicação então pode entrar em modo urgente e fazer com que os dados enviados no segmento em questão sejam

Tabela II.2: Significado dos bits do campo CODE BITS.

BIT(da esquerda p/ direita)	SIGNIFICADO (se o bit for 1)
URG	Campo URGENT POINTER é válido
ACK	Campo ACKNOWLEDGEMENT é válido
PSH	Este segmento é repassado imediatamente para a aplicação, não sendo armazenado no buffer
RST	Reinicia a conexão
SYN	Sincroniza os números de seqüência
FIN	O emissor alcançou o fim de sua *stream* de bytes

processados tão logo recebidos, e antes do que qualquer dado que porventura já tenha sido recebido e esteja armazenado no *buffer* de recepção. Este mecanismo é útil principalmente em aplicações de terminais remotos, onde o usuário pode querer interromper um processo após o envio de muitos bytes, mas sem querer esperar pelo processamento de todos estes bytes para que haja a interrupção. O bit URG quando colocado em 1 indica portanto o envio de dados urgentes e é utilizado em conjunto com o campo URGENT POINTER (ponteiro urgente), o qual indica o byte onde os dados urgentes terminam.

O bit ACK (reconhecimento) indica que o campo ACKNOWLEDGEMENT NUMBER é válido, ou seja, o segmento está sendo utilizado para reconhecer dados recebidos.

O bit PSH (*push* — empurrar) faz com que os dados sejam entregues à aplicação assim que sejam recebidos sem esperar pelo enchimento do *buffer* de recepção. Este esquema é muito útil em aplicações interativas, onde o usuário requer tempos de resposta muito curtos. Num terminal remoto por exemplo, a aplicação na fonte enviaria um segmento com apenas um byte referente a cada tecla pressionada pelo usuário colocando o valor 1 no bit PSH.

Os demais bits RST, SYN e FIN se referem ao estabelecimento e término de conexões e serão analisados na seção II.5.

O campo WINDOW com 16 bits, representa o tamanho, em bytes, da janela anunciada pelo receptor, a *advertised window*. Como o tamanho é definido por 16 bits, o limite do valor da *advertised window* é de 65536 bytes. Esse valor máximo traz uma limitação no desempenho do TCP [10]. Em enlaces com produto "retardo-banda passante" muito grandes, mesmo que o receptor tenha espaço suficiente em *buffer* para manter a janela em seu valor máximo, os 65536 bytes podem não ser suficientes para que o canal se mantenha completamente ocupado. A tabela II.3 apresenta os tamanhos de janela necessários para que se obtenha uma ocupação completa de um enlace com RTT de 100 ms, que representa tipicamente uma conexão continental. Como se pode notar, já em um enlace T3, a limitação do campo WINDOW não permite uma ocupação completa do meio. O IETF (*Internet Engineering Task Force*) já está trabalhando em uma extensão para o TCP que permita o aumento do tamanho da janela anunciada.

Tabela II.3: Tamanho necessário da janela de transmissão para RTT de 100 ms.

Banda Passante	Produto Retardo X Banda
T1 (1,5 Mbps)	18 KB
Ethernet (10 Mbps)	122 KB
T3 (45 Mbps)	549 KB
FDDI (100 Mbps)	1,2 MB
STS-3 (155 Mbps)	1,8 MB
STS-12 (622 Mbps)	7,4 MB
STS-24 (1,2 Gbps)	14,8 MB

Por último o campo CHECKSUM é utilizado para verificar a integridade tanto dos dados quanto do cabeçalho. O processo é feito da mesma forma que no protocolo UDP, onde um pseudo cabeçalho e zeros são adicionados ao segmento para que todo ele seja múltiplo de 16 bits. Vale frisar que nem o pseudo cabeçalho nem os zeros adicionados são transmitidos. Após este passo, o software TCP computa o complemento a 1 da soma dos complementos a 1 de todos os blocos de 16 bits do segmento alterado. Neste passo o campo CHECKSUM será considerado como contendo somente zeros. No nó destino o software TCP executa o mesmo processo e descarta o pacote em caso de erro.

O formato do pseudo cabeçalho de 12 bytes (figura II.6) contém os endereços IP fonte e destino (campos SOURCE IP ADDRESS e DESTINATION IP ADDRESS) que servem para verificar se o segmento chegou ao destino correto. O campo PROTOCOL indica qual o valor do protocolo que a camada inferior irá utilizar no campo de tipo de protocolo (campo PROTOCOL TYPE no datagrama IP) e assume o valor 6 (seis) para o TCP. O campo TCP LENGTH especifica o tamanho total do segmento incluindo o cabeçalho.

0	8	16	31
SOURCE IP ADDRESS			
DESTINATION IP ADDRESS			
ZERO	PROTOCOL	TCP LENGTH	

Figura II.6: Formato do pseudo cabeçalho.

Como já foi visto, nada impede que o tamanho de um segmento TCP possa variar ao longo de uma conexão. Porém, deve haver um acordo sobre o tamanho máximo que um segmento pode assumir durante uma conexão. Para isso, o software TCP em ambos os extremos de uma conexão mantém um valor denominado MSS - *Maximum Segment Size* limitando o tamanho do segmento.

A escolha do valor MSS tem influência direta no desempenho do protocolo TCP na medida em que valores muito pequenos acarretarão um uso ineficiente da largura de faixa da rede, e valores muito altos podem fazer com que o datagrama IP seja fragmentado ao longo do percurso. Esta fragmentação obrigará a espera de todos os fragmentos para que o software TCP possa enviar o reconhecimento (fragmentos não podem ser reconhecidos em separado). Logo deve-se procurar um ponto ótimo.

Embora importante, a determinação deste valor ótimo nem sempre é simples. Dentro de uma rede local este problema se torna mais simples, sendo possível usar um valor que resulte em um datagrama IP com tamanho igual ao MTU (*Maximum Transfer Unit*) da rede. Caso o caminho envolva várias redes heterogêneas pode-se tentar descobrir o tamanho ideal ou escolher o valor 536 bytes, o qual resultará em um datagrama IP de tamanho padrão (576 bytes) [11].

II.5 Estalebecimento e Término de uma Conexão

Conforme já citado anteriormente, o protocolo TCP é orientado a conexão, o que significa que para haver troca de dados primeiramente os dois pontos devem concordar em estabelecer uma conexão TCP.

Para que isto ocorra, um ponto-final deve executar uma função de abertura passiva (*passive open*), sinalizando ao sistema operacional que está apto a estabelecer uma conexão com quem porventura desejar. O sistema operacional retorna então o número da porta que será utilizada por aquela aplicação na conexão em questão. O outro ponto-final deve então executar uma função de abertura ativa (*active open*) para que a conexão possa ser completada.

Mais detalhadamente, o processo utiliza um mecanismo denominado *three-way handshake*:

1. O ponto que executou a função de abertura ativa envia um segmento com o bit SYN (*synchronization*) do campo CODE BITS (ver seção II.4) com valor 1, número da porta destino igual à porta do outro ponto com a qual deseja se conectar e número de seqüência inicial (ISN - *Initial Sequence Number*) x.

2. O ponto que espera pela abertura da conexão passivamente responde então com um segmento com o bit SYN com valor 1, número da porta destino igual à porta que o outro ponto informou como porta fonte no segmento recebido, número de seqüência inicial y e número de reconhecimento igual a $x + 1$.

3. O ponto que iniciou o processo (passo 1) envia um segmento com número de reconhecimento igual a $y + 1$, reconhecendo portanto o segmento recebido no passo 2.

Este segmento não possui o bit SYN com valor 1. Este passo é utilizado apenas para indicar que os dois lados concordam que a conexão foi estabelecida.

O processo é ilustrado na figura II.7.

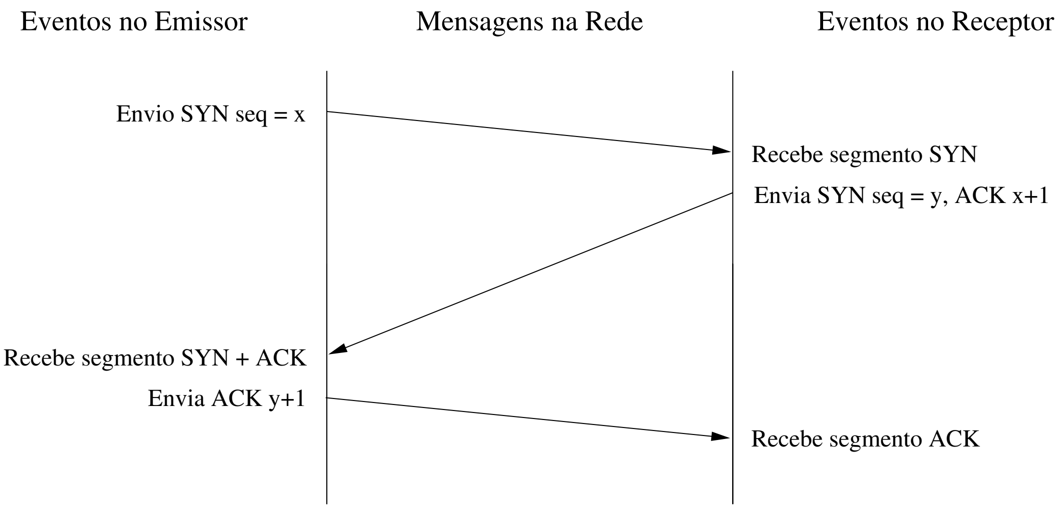


Figura II.7: Estabelecimento de uma conexão TCP.

Este mecanismo pode funcionar tanto quando a conexão é estabelecida em um dos sentidos quanto em ambos. No último caso, as conexões podem ser estabelecidas simultaneamente e os dados trafegarão de forma independente nos dois sentidos.

Se para a abertura de conexões nos dois sentidos o processo é simultâneo, para o fechamento da conexão cada ponto pode decidir fazê-lo quando não houver mais dados a serem enviados. Os passos para que um ponto-final feche a conexão em um dos sentidos estão descritos a seguir:

1. O nó fonte, após enviar o último segmento de dados, espera que o nó destino reconheça todos os dados.

2. O nó fonte envia um segmento com o bit FIN (*finished*) com valor 1.

3. O nó destino por sua vez reconhece o segmento recebido e informa à aplicação do seu lado que nenhum dado a mais será recebido (fim de arquivo por exemplo).

4. O nó fonte recebe o reconhecimento.

A figura II.8 ilustra o processo, mostrando o encerramento da conexão em ambos os sentidos. Esta é a forma normal de se terminar uma conexão. Porém, em certas situações uma aplicação pode ter que interromper uma conexão imediatamente. Para fazê-lo basta enviar um segmento com o bit RST (*reset*) com o valor 1. O outro nó imediatamente considera a conexão fechada e informa à aplicação. Neste caso, a conexão é terminada nos dois sentidos e os *buffers* são liberados.

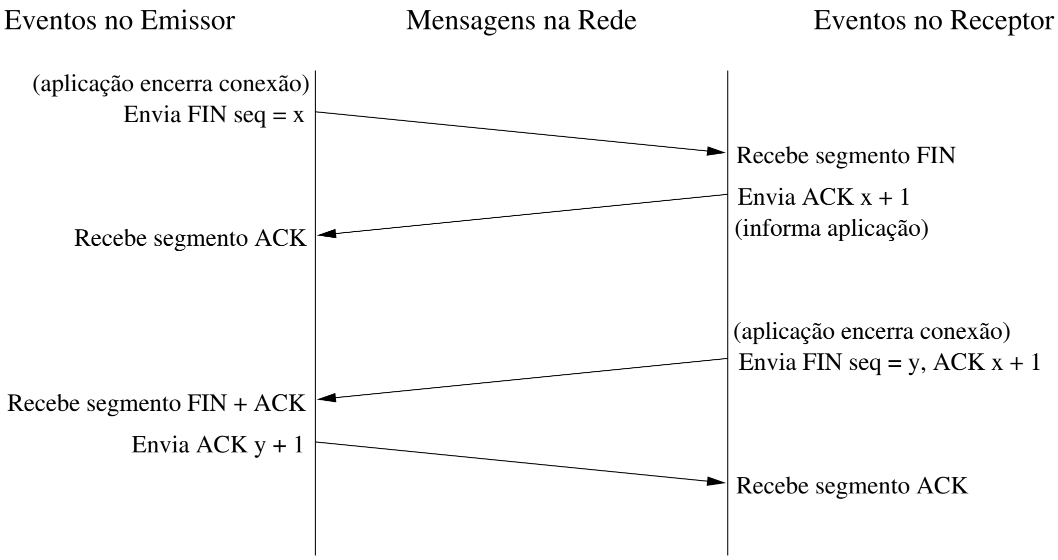


Figura II.8: Término de uma conexão TCP nos dois sentidos.

Como já citado na introdução deste capítulo, buscou-se aqui uma breve apresentação do TCP. O próximo capítulo detalha um aspecto de extrema importância para o funcionamento do protocolo, o controle de congestionamento, que é o foco principal deste trabalho.

Capítulo III

Controle de Congestionamento do TCP

O congestionamento nas redes de computadores é um problema que não pode ser resolvido com o simples aumento na disponibilização de recursos. Este fato é explicado no início deste capítulo. Após a verificação de que são necessários mecanismos de controle de congestionamento em uma rede, são apresentados os esquemas de temporização, de ocupação do canal e de resposta ao congestionamento desenvolvidos para o TCP. Melhorias ao TCP como a proposta de reconhecimentos seletivos são explicadas em seguida. No final do capítulo algumas versões mais conhecidas de implementações do protocolo TCP são apresentadas, descrevendo-se os conjuntos de mecanismos de controle de congestionamento utilizados por cada uma delas.

III.1 Congestionamento: Entendendo o Problema

Pode-se definir congestionamento como uma situação em que a demanda pelos recursos da rede se aproxima ou ultrapassa a sua capacidade [12]. A função básica do controle de congestionamento é a alocação de recursos de modo que a rede opere da forma mais eficiente possível nessas situações. Entre os recursos de uma rede, podemos citar: a largura de banda dos enlaces, espaço em *buffers* e tempo de processamento nos nós intermediários da rede.

Ao contrário do que se poderia pensar, a solução para tais situações não está em simplesmente aumentar a capacidade da rede. Tomemos como exemplo os *buffers* dos nós de uma rede. Quando a memória não é suficiente para acomodar o tráfego, ocorre um transbordo e conseqüentemente, perda de pacotes. Mas o que aconteceria se tivéssemos buffers infinitos? Se a fila nos nós não transbordasse, em situações de congestionamento ela aumentaria tanto o retardo na transmissão que isto geraria *time-outs* nos transmissores, que retransmitiriam os pacotes enfileirados (dados como perdidos). Obviamente isto iria piorar a situação. Quando o congestionamento fosse contornado e os pacotes chegassem aos seus respectivos destinos, haveria duplicidade e muitos seriam descartados depois de terem ocupado desnecessariamente os recursos da rede.

Outro bom exemplo disso é o aumento da velocidade dos enlaces. Intuitivamente, poderíamos pensar que esta seria uma solução. No entanto, a inserção de enlaces de alta velocidade em uma rede não implica no descarte dos enlaces antigos, mais lentos. Isto gera uma disparidade que por si só causa congestionamento. Se um roteador recebe um *stream* de dados a 2Mbps e tem de escoá-lo por um enlace de 64kbps, é óbvio que em situações de demanda mais alta haverá transbordo dos buffers. Além disso, mesmo que só haja enlaces de alta velocidade, a topologia da rede pode torná-la suscetível a congestionamento. Em particular, as redes IP têm uma tendência a sobrecarregarem a melhor rota entre dois pontos enquanto que as demais ficam subutilizadas. Esses são problemas crônicos em uma rede heterogênea como a Internet, em que se tem que lidar com enlaces intermediários que vão desde linhas assíncronas a 14.4kbps a troncos óticos de 155Mbps e maiores e uma topologia sem um controle centralizado.

Como o congestionamento é um problema que consiste na demanda por recursos exceder sua disponibilidade, a priori teríamos duas abordagens para resolvê-lo: aumentar dinamicamente os recursos disponíveis ou reduzir a demanda para que ela se ajuste aos recursos presentes. A primeira solução, se considerada isoladamente, pode somente adiar o problema, pelas razões já vistas anteriormente. Assim, devemos nos concentrar nos esquemas de redução da demanda. Estes podem ser divididos em duas categorias: bloqueio de serviço e degradação de serviço. O primeiro não permite que novas sessões sejam iniciadas quando a rede está saturada. Exemplos de redes que empregam esse esquema são a rede telefônica e redes de dados orientadas a conexão, como o ATM, que possui um mecanismo de controle de admissão (CAC - *Connection Admission Control*). Por outro lado, a abordagem de degradação de serviço tenta fazer com que os usuários reduzam a carga que estão colocando na rede. Este tipo de solução se adequa a redes de pacotes sem conexão, onde não se pode negar uma nova sessão. Redes orientadas a conexão também podem usar esquemas de degradação, combinando-os com os de bloqueio. Isso é feito, por exemplo, no ATM.

O controle de congestionamento necessariamente envolve algum tipo de realimentação que indique aos nós ou aos usuários a ocorrência desta condição, para que estes tomem as providências adequadas. Existe uma série de maneiras de prover esta realimentação:

- Mensagens: O nó onde ocorre o congestionamento envia mensagens explícitas aos outros nós e às fontes de tráfego. Essas mensagens são chamadas *choke packets*.

- Campos indicadores nos pacotes: Ao invés de mensagens, a rede usa campos nos pacotes de dados para indicar sobrecarga. O nó sobrecarregado muda o valor de um flag no pacote. Este flag pode indicar congestionamento no caminho percorrido pelo próprio pacote (neste caso a informação precisa ser redirecionada de volta à fonte) ou no caminho inverso (já é direcionado para a fonte de tráfego excessivo). Redes Frame Relay e ATM empregam este esquema de realimentação.

- Notificação implícita: Nesta solução, não há indicações. A rede simplesmente descarta os dados em excesso, esperando que isso gere uma diminuição da velocidade do transmissor. Isso é feito no IP, de acordo com sua característica de melhor esforço, e espera-se que o TCP detecte esta condição e reduza o ritmo.

Outra questão é onde devem ser implementados os mecanismos de realimentação e de controle. O congestionamento ocorre na rede, portanto os nós intermediários são os pontos mais indicados para gerar a realimentação. Os mecanismos que realizam esta funcionalidade devem ser incorporados ao protocolo de rede. Analogamente, como o tráfego é gerado pelos *hosts*, eles são naturalmente o ponto onde se pode melhor ajustar a carga que é colocada sobre a rede. Assim, a camada transporte adquire grande importância como ponto de controle.

As ações a serem tomadas pela camada rede são no sentido de restringir o serviço aos usuários, negando-o ou degradando-o. Esta é uma ação de controle, aliviando o congestionamento no nó, e pode ser também de realimentação, pois indica à camada transporte a ocorrência de congestionamento. Os mecanismos de descarte podem ou não ter algum tipo de inteligência com relação ao controle de tráfego. O *leaky bucket* usado no ATM exerce a função de policiamento de tráfego. As células que excederem o contrato do usuário são descartadas primeiro caso haja congestionamento. Neste caso a ação de controle também parte da camada rede.

Assim, vemos que há uma grande variedade de soluções para o controle de congestionamento. Este campo tem se mostrado bastante vasto para a pesquisa, e ainda está longe de ser esgotado. Existem duas razões para isso: os requisitos a que um esquema deve atender são complexos e muitas vezes conflitantes, o que força compromissos e deixa espaço para otimizações; além disso, os esquemas de controle de congestionamento são muito específicos com relação às características da rede onde são aplicados. Os mecanismos criados tendo em mente uma determinada arquitetura de rede não necessariamente serão eficazes com outra. Por exemplo, no Frame Relay e no ATM são usadas notificações para sinalizar que o caminho por onde o pacote ou célula passou está sobrecarregado. No IP, os pacotes são roteados de forma independente, portanto essa indicação não possui qualquer utilidade.

Entre os requisitos aos quais um esquema de controle de congestionamento deve atender, podemos enumerar:

- Eficiência ou baixo *overhead*: Um dado esquema de controle de congestionamento não deve gerar muito tráfego de controle, pois isso reduz a banda-passante útil da rede. Isso é ainda mais crítico quando a rede está congestionada. Por exemplo, um nó pode gerar mensagens para informar aos *hosts* e demais nós que a rede está saturada. No entanto, isto consome recursos da rede justamente quando eles estão mais escassos.

- Baixo tempo de resposta: As alterações na demanda e na capacidade da rede são muito dinâmicas. Portanto, o protocolo deve fazer com que os usuários diminuam sua demanda prontamente quando houver diminuição da disponibilidade de recursos, e ao mesmo tempo notificá-los que podem aumentar a demanda quando a disponibilidade aumentar. Um esquema baseado em notificações de congestionamento não é o mais eficaz em redes com enlaces de grande latência. Até que a mensagem avisando da saturação chegue aos hosts, já poderão ter sido transmitidos (e estarão armazenados no meio) uma quantidade de dados igual ao produto banda-atraso do enlace. Se tivermos um STM-1 (155Mbps) com 60ms de atraso, isso significa mais de 9Mbits despejados na rede após a emissão da mensagem. Portanto, a rede tem que detectar a ocorrência do congestionamento no seu início e tomar as ações necessárias para evitar que ele chegue a níveis mais críticos.

- Robustez: O mecanismo deve ser capaz de funcionar face a condições adversas. Durante as situações de congestionamento, as taxas de perdas de pacotes, erros, pacotes fora de ordem etc. aumentam muito. Portanto, notificações de congestionamento podem não chegar ao seu destino justamente nestes momentos em que o controle de congestionamento é mais necessário.

- Otimização da vazão total da rede: Isto se refere à otimização do desempenho da rede como um todo. É importante que os mecanismos de controle de congestionamento não reduzam o serviço de tal forma que a rede fique subutilizada. A eficiência no aproveitamento dos recursos da rede é fundamental, especialmente quando a demanda é alta.

- Respeito às políticas da rede: Se a rede tem como premissa a igualdade entre os usuários, uma situação de congestionamento não pode fazer com que um usuário fique sem serviço enquanto um outro pega toda a banda disponível. No entanto, a proposta da rede pode envolver tratamento diferenciado para usuários ou aplicações. Isso é necessário para que a rede transporte tipos diferentes de tráfego (tráfego sensível a atraso, como voz, ou sensível a perdas, como transferência de arquivos). Um exemplo são as filas com múltiplas prioridades de descarte e de emissão, que permitem tratar o tráfego de acordo com sua necessidade. A implementação de mecanismos que assegurem qualidade de serviço é sempre uma solução de compromisso. Se um usuário possui serviço garantido, pode ser que outros tenham que ser deixados sem serviço algum para que se atenda ao primeiro, gerando demanda reprimida. Claramente isto agrega complexidade ao problema.

Como foi dito, uma solução de controle de congestionamento tem de levar em consideração as particularidades da arquitetura da rede em que ela será implementada. As redes IP não são orientadas a conexão, e provêem um serviço do tipo "melhor esforço". Caso haja sobrecarga, descartem-se os pacotes. Os roteadores IP originalmente possuiam filas FIFO, ou seja, os pacotes eram descartados indiscriminadamente a partir do momento em que o *buffer* se esgotava. Mais recentemente, têm sido implementados mecanismos que monitoram o tamanho da fila e começam a descartar pacotes antes de esgotado o *buffer*, agindo de forma mais preventiva. Entre esses está o RED (*Random Early Discard*) [13], juntamente com todas as suas variações. Além disso, a arquitetura de Serviços Diferenciados [14] [15] propõe políticas de descarte específicas para cada classe de serviço.

Vale ressaltar que a eficácia dos mecanismos de descarte está intimamente ligada à maneira como a camada transporte trata as perdas de pacotes. Portanto, para analisar o impacto da incorporação desses mecanismos ao protocolo IP é necessário compreender como o TCP detecta o congestionamento e como ele trata cada situação. O controle de congestionamento fim-a-fim implementado no TCP é, portanto, o escopo deste trabalho.

III.2 Mecanismos de Controle de Congestionamento do TCP

O controle de congestionamento do TCP tem como princípio básico o fato de que o fluxo de pacotes na rede deve ser conservativo. Isto implica em que um pacote só pode ser colocado na rede depois que um pacote for retirado. O recebimento de um ACK é o que indica para o transmissor que um pacote deixou a rede. Como o transmissor só pode enviar mais segmentos ao receber ACKs, o protocolo se torna *self-clocking*, ou seja, o protocolo ajusta automaticamente sua taxa de transmissão ao enlace mais lento da rede. Isto pode ser visualizado no esquema da figura III.1. Essa característica é essencial para lidar com a heterogenidade da Internet.

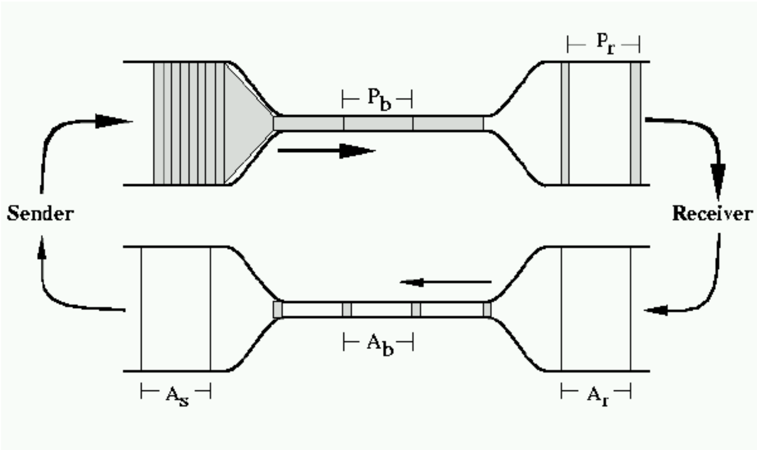


Figura III.1: Mecanismo de *self-clocking* utilizado pelo TCP.

A figura III.1 é uma representação do mecanismo de *self-clocking*. Ela apresenta duas redes de alta velocidade interligadas por um enlace de gargalo, mais lento e com maior atraso. A dimensão horizontal é o tempo e a vertical é a banda passante. A área de cada retângulo equivale então ao produto Tempo x Banda-passante, que é a quantidade de bits enviados. Assim no enlace mais lento, esses bits demoram mais para serem transmitidos, levando um tempo P_b, como mostrado. Ao chegar no enlace mais veloz, do receptor, os pacotes passam a estar espaçados pelo tempo entre as chegadas, que vai ser o mesmo tempo P_b. Desta forma os pacotes de reconhecimento serão gerados num espaçamento de tempo também igual a P_b. Como os pacotes de dados são maiores que os respectivos reconhecimentos, esse espaçamento será mantido no enlace de gargalo, sendo A_b=P_b. Como novos pacotes só serão enviados com o recebimento dos reconhecimentos, e a taxa deste recebimento é igual a taxa no enlace mais lento da rede, fica demonstrado que a taxa do emissor será igual a taxa do enlace mais lento, no caso, o de gargalo.

A condição acima vale para uma conexão em estado estacionário. Há portanto a necessidade de um algoritmo que determine o comportamento transiente do TCP, desde o início da conexão até que se atinja o equilíbrio. Este algoritmo deve ser agressivo o suficiente para atingir o ponto de equilíbrio o mais rápido possível e, ao mesmo tempo, conservador o suficiente para manter a estabilidade da rede, evitando que se inunde as filas dos nós intermediários com rajadas de dados muito maiores do que elas possam absorver.

Mesmo com todos estes algoritmos para evitar o congestionamento, a ocorrência destes é inevitável. Assim, o protocolo deve prover meios de se detectá-los com rapidez e precisão. O congestionamento deve ser percebido através de um ou mais parâmetros aos quais o transmissor tenha acesso. O parâmetro usado pelo TCP é a perda de pacotes. Como as redes atuais apresentam baixas taxas de erro binário, pode-se supor com segurança que quando ocorre uma perda, ela é causada por congestionamento na rede.

O principal indicador que o TCP usa para inferir que houve uma perda é o tempo entre a transmissão de um pacote e a chegada do reconhecimento relativo àquele pacote. Como o tempo necessário para o pacote chegar ao receptor não é conhecido a priori, ele tem de ser estimado no início e no decorrer da conexão. Uma conexão TCP pode envolver desde *hosts* em uma mesma rede local até *hosts* separados por enlaces de satélite, portanto a estimativa tem de abranger uma grande faixa de valores mantendo uma boa precisão.

O controle de congestionamento do TCP é realizado por quatro algoritmos: *slow start, congestion avoidance, fast retransmit e fast recovery.* Apesar de serem independentes, esses algoritmos são geralmente implementados de forma conjunta. Outro aspecto do controle de congestionamento do TCP é a estimativa do RTT (*Round Trip Time*), que adquire importância na medida em que a partir dela se estabelece o valor do temporizador que determina as retransmissões de dados. A precisão da estimativa do RTT é importante porque se ele for subestimado, o TCP fará retransmissões desnecessárias. Se por outro lado ele for superestimado, ele levará muito tempo para realizar retransmissões, fazendo com que o protocolo perca em eficiência.

III.2.1 Temporização

O cálculo da temporização para retransmissão no TCP é sem dúvida um fator de extrema importância para o funcionamento eficiente do protocolo. A correta estimação do tempo de envio de um pacote permite que o tempo de espera por um sinal de reconhecimento seja o mínimo necessário. Isto garante que em caso de perda de dados o transmissor não fique em um estado de espera ociosa, não efetuando as retransmissões necessárias.

Já no caso contrário, o transmissor não deve subestimar o tempo necessário para o envio de um pacote e recebimento do sinal de reconhecimento. Se isto ocorrer, ou seja, se o reconhecimento chegar à fonte de dados após a expiração do temporizador, o TCP irá interpretar uma perda, exigindo a retransmissão de dados além de ativar os mecanismos de controle de congestionamento, que irão reduzir a taxa de transmissão. Assim, o emissor estará reenviando dados que já foram transmitidos com sucesso, desperdiçando banda passante e tempo.

Desta forma o temporizador de retransmissão deve ser capaz de absorver as variações no tempo necessário para o tráfego dos dados na rede, se adaptando, de modo a estimar da maneira mais acurada o temporizador de retransmissão.

Round Trip Time (RTT)

O RTT é o tempo que um pacote necessita para ir e voltar da fonte de dados ao destino. Esta variável depende do tempo de propagação do sinal no meio de transmissão e do tempo gasto pelos dados nas filas dos roteadores. Desta forma, o RTT varia de acordo com a carga a qual a rede está sendo submetida. Quanto maior o tráfego na rede, maiores serão as filas nos roteadores e maiores serão os tempos de espera. Em geral, para redes de pacotes, a medida do RTT possui altos desvios padrões.

Estas grandes variações nos valores do RTT tornam a estimação acurada do temporizador de retransmissão uma tarefa não trivial. A estimativa do RTT é um fator importante no cálculo da temporização.

A especificação original do TCP realizava a estimativa do RTT de forma simples, como uma média ponderada entre amostragens da variável [16]. Assim, para se estimar o valor atual do RTT, utiliza-se os valores antigos deste parâmetro. A cada pacote enviado, o tempo entre a transmissão dos dados e o recebimento do reconhecimento é armazenado. O valor do RTT estimado é então calculado como uma média ponderada entre os valores antigos e o último RTT amostrado. A equação III.1 mostra o cálculo descrito.

$$RTT_{estimado} = (\alpha * RTT_{anterior}) + [(1-\alpha) * RTT_{ultima\_amostragem}] \tag{III.1}$$

Na equação, podemos verificar que a variável $RTT_{estimado}$ atual irá receber o último valor de RTT amostrado ($RTT_{ultima\_amostragem}$) somado ao valor anteriormente utilizado ($RTT_{anterior}$). A soma é ponderada através do fator α. A referência [3] utiliza α tipicamente igual a 0,9, o que torna a estimativa do RTT praticamente imune à grandes variações em um único pacote, dando maior peso à média antiga, que já levou em conta uma grande quantidade de amostras do RTT.

Na especificação original do TCP o cálculo do temporizador de retransmissão era feito diretamente sobre o valor estimado para o RTT. Assim a equação III.2 inclui o parâmetro β que ajusta o tempo de espera pelo reconhecimento. A escolha do parâmetro influencia diretamente a eficiência do TCP, realizando uma solução que tem um compromisso entre espera ociosa e retransmissões desnecessárias de pacotes, como já foi explicado anteriormente.

$$Timeout = \beta * RTT \tag{III.2}$$

O valor de β sugerido para as primeiras implementaçoes do TCP foi de 2. Trabalhos posteriores mostram entretanto outras técnicas para a estimação do valor de *timeout*.

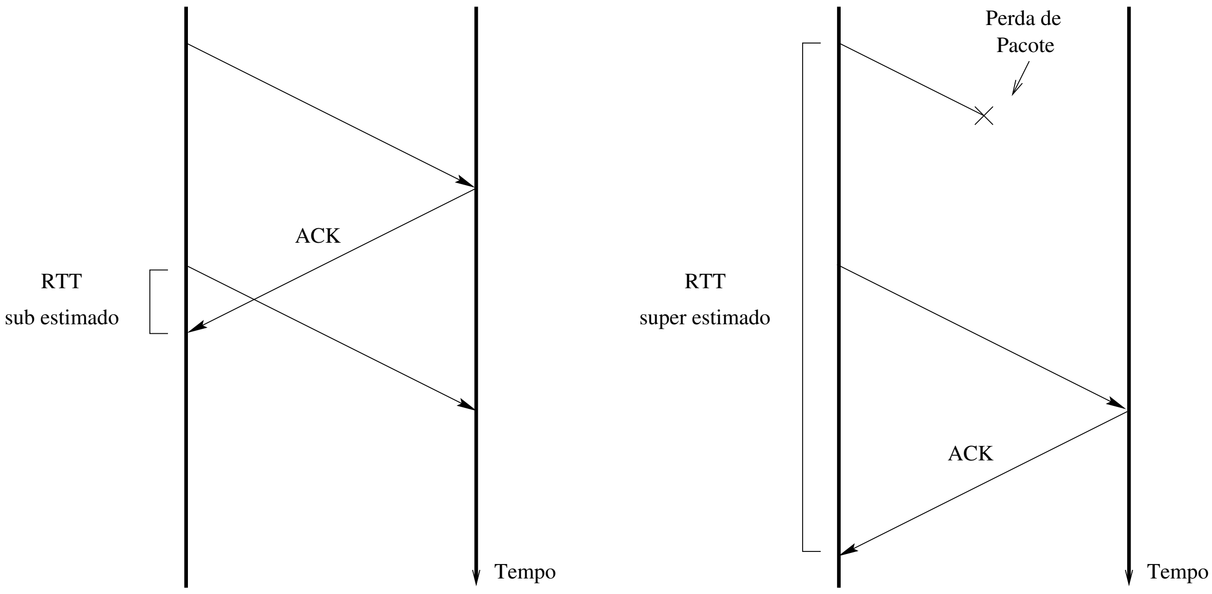
Técnicas de Medida do RTT

A medida do RTT não pode ser realizada de maneira trivial como foi descrito anteriormente. Não se consegue boas estimativas do RTT simplesmente enviando um datagrama e aguardando um sinal de reconhecimento. Deve ser lembrado que este sinal se refere ao reconhecimento de um conjunto de dados, não havendo um vínculo com determinado datagrama. Este fato acarreta no surgimento de problemas nas estimativas realizadas, quando ocorrerem retransmissões sem que perdas tenham ocorrido. Nestes casos, não se sabe a qual transmissão (a primeira tentativa ou a retransmissão) o sinal de reconhecimento (ACK) se refere. Este fato é conhecido como ambigüidade de reconhecimento.

Podemos utilizar um exemplo para explicar melhor o problema. Suponhamos que houve uma situação de congestionamento onde o RTT aumentou drasticamente o seu valor. Isto faria com que o próximo valor de RTT fosse mal estimado. Assim, ao efetuar a transmissão de dados, o emissor poderia atingir o limite do temporizador, forçando uma retransmissão dos dados, mesmo que não tenham ocorrido perdas. Uma situação possível seria que logo após o reenvio dos dados o sinal de reconhecimento destes, referente à primeira transmissão, chegasse no emissor. Se este interpretasse este sinal como o reconhecimento da retransmissão, o RTT ficaria subestimado. Como a estimativa do próximo valor do RTT é influênciada pelo valor atual, aquela também ficaria menor que o real. Desta forma, a situação iria piorar cada vez mais. A figura III.2 (a) ilustra este fato.

Existem também situações onde o inverso ocorre [17]. O emissor considera o sinal de ACK como sendo da primeira tentativa de transmissão, quando na realidade este referenciava a retransmissão, como mostrado na parte (b) da figura III.2. Nestes casos o RTT estimado pode crescer indefinidamente quando ocorrem perdas na rede.

Uma solução simples para o problema da estimação de RTT quando ocorrem retrans-



(a) (b)

Figura III.2: Estimativa do RTT em caso de retransimssão quando se utiliza como referência: (a) a retransmissão, ou (b) o primeiro pacote enviado.

missões é conhecida como algoritmo de Karn/Partridge. Este algoritmo utiliza apenas os sinais de ACK que não são ambíguos para a realização da estimativa do RTT. Assim esta variável somente é estimada quando não há retransmissões. Além disto, é necessário que se implemente uma política que aumente gradativamente o temporizador quando ocorrem estouros de temporização (*timeouts*), para que o TCP possa lidar com os aumentos rápidos do RTT. A estratégia de aumento multiplicativo do temporizador em caso de *timeout* é a mais comumente utilizada nas implementações do TCP a fim de resolver este problema. O valor típico do fator multiplicativo é 2.

O maior problema das técnicas apresentadas acima, que fizeram parte da especificação original do TCP é a dificuldade de se adaptar às variações bruscas que podem ocorrer no RTT de uma rede submetida a grandes cargas. Pode ser provado [17] que o fator β utilizado como descrito na equação III.2 e com valor de 2 só consegue atender variações do RTT para uma rede com carga em 30% da capacidade máxima.

Uma forma de melhorar as estimativas do RTT foi proposta por [3]. É sugerido que se substitua a constante β, passando a se utilizar o desvio padrão do RTT no cálculo do temporizador.

Para simplificar a implementação, utiliza-se nos cálculos o desvio médio em vez do desvio padrão do RTT. As equações III.3 descrevem o cálculo do temporizador.

$$DIF = RTT_{atual} - RTT_{antigo} \tag{III.3}$$

$$RTT_{suavizado} = RTT_{antigo} + \delta * DIF$$

$$DEV = DEV_{antigo} + \rho(|DIF| - DEV_{antigo})$$

$$Timeout = RTT_{suavizado} + \eta * DEV$$

onde DEV é o desvio médio estimado, δ é um fator entre 0 e 1 que controla o peso com que as amostras novas afetam a média ponderada, ρ, também entre 0 e 1 indica o peso com que uma nova amostra afeta o desvio médio e η regula quanto o desvio médio afeta a estimativa do temporizador. Os valores das constantes δ e ρ são preferencialmente dados em inversos de potências de 2 para tornar a computação eficiente. Pesquisas sugerem valores de ρ=1/8 e δ=1/4 e η=3.

Apesar de toda a sofisticação para o cálculo do temporizador, um fator que irá limitar sua precisão será a granulosidade do relógio utilizado para amostrar os RTTs. Como exemplo pode ser citado o caso típico do TCP de um sistema *Berkeley Unix*, onde a granulosidade do relógio pode ser de até 500 ms. Isso faz com que a qualidade da estimativa do RTT fique prejudicada, uma vez que em enlaces continentais algumas vezes os RTTs não chegam à 200 ms. Para piorar o cenário, implementações do TCP para alguns sistemas Unix só checam a ocorrência de estouros de temporização a cada 500 ms, o que piora ainda mais a estimativa do temporizador de retransmissão.

III.2.2 Slow Start

As implementações mais antigas do TCP iniciavam suas conexões enviando uma rajada de segmentos do tamanho da janela anunciada pelo receptor. Este comportamento, insensível às condições de rede, tendia a causar sobrecarga nas filas dos roteadores, que com isso perdiam pacotes.

A perda de pacotes era percebida pelo TCP através do estouro de uma temporização, que então verificava o último segmento reconhecido pelo receptor e enviava outra rajada, partindo do segmento seguinte a ser enviado. Assim, o transmissor continuava a colocar rajadas de pacotes em uma rede que já estava congestionada, piorando a situação. Este comportamento tornava as redes altamente instáveis, pois não havia um mecanismo que limitasse o envio de pacotes pelo transmissor em função destes *time-outs*. O exemplo concreto desta instabilidade eram os "colapsos de congestionamento" citados por Jacobson [3], em que se conseguia uma vazão de apenas 40 bps em um enlace de 32 Kbps.

Como já foi dito, deve existir algum mecanismo que controle o comportamento transitório da conexão levando-a ao ponto de equilíbrio de forma rápida, que entretanto não cause congestionamento na rede. A solução proposta em [3] para este problema foi iniciar-se a conexão com uma janela pequena, aumentando-a de forma gradual até que se atinja a capacidade da rede. Isto se aplica tanto a uma conexão que se inicia como ao reinício após um estouro do temporizador, quando o congestionamento já existe e se deseja esvaziar o meio e começar novamente a manipulação da janela de transmissão.

A implementação desta solução é feita da seguinte forma:

1. Adiciona-se a variável *congestion window (cwnd)*;

2. No início da conexão, *cwnd* recebe o valor 1*MSS;

3. A cada novo ACK (não duplicado [1]) recebido pelo transmissor, *cwnd* é incrementada

[1] Como a política de reconhecimentos é cumulativa, o receptor confirma o recebimento de todos os pacotes até o indicado no ACK. Assim segmentos recebidos fora de ordem geram ACKs duplicados. A seção III.2.4 irá detalhar os ACKs duplicados.

de 1*MSS;

4. A janela de transmissão é a menor entre a janela anunciada pelo receptor (*awnd*) e *cwnd*.

O procedimento de aumentar *cwnd* de um pacote para cada ACK recebido resulta em crescimento exponencial da janela. Isto pode ser visto na figura III.3, sendo compreendido da seguinte forma:

- O transmissor inicia a conexão, com $cwnd = 1$*MSS, e envia um pacote;
- Passado um RTT, o transmissor recebe o ACK deste pacote, e incrementa *cwnd* de um pacote ($cwnd = 2$). Como um ACK foi recebido, o transmissor pode enviar mais dados (os ACKs são o *clock* do transmissor) e envia dois pacotes;
- Depois de um RTT, o transmissor recebe dois ACKs, incrementando *cwnd* de dois pacotes ($cwnd = 4$). Da mesma forma que antes, o transmissor agora envia uma rajada de quatro pacotes, e assim por diante.

Assim, a janela aumenta exponencialmente. É claro que em um dado momento se atinge a capacidade da rede, pois rajadas cada vez maiores exigem espaço nas filas dos roteadores. Do ponto de vista do *slow-start*, a única ação a ser tomada com relação a isto é que quando ocorrer um *time-out*, *cwnd* volta a 1*MSS e o algoritmo é reiniciado. Outros algoritmos que veremos mais adiante tratam da adaptação do transmissor às limitações de banda-passante na rede. O objetivo do *slow start* é permitir que uma conexão iniciada ou reiniciada após um *time-out* atinja rapidamente o seu ponto de equilíbrio.

III.2.3 Congestion Avoidance

Conforme foi dito na seção anterior, o *slow start* incrementa a janela de transmissão exponencialmente. Em um determinado ponto, a capacidade da rede será ultrapassada, provocando perdas de pacotes. Isto indica que a janela de congestionamento ultrapassou

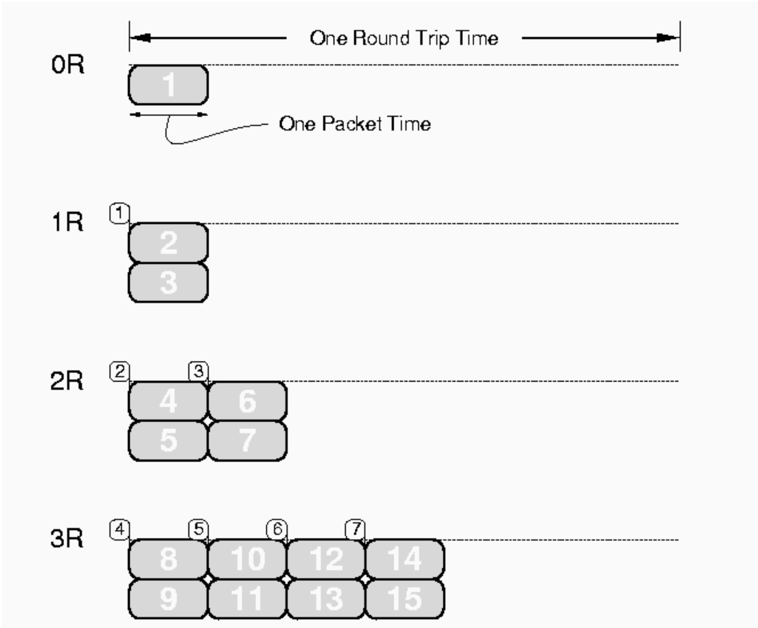


Figura III.3: Crescimento exponencial da janela na fase de *slow-start*.

o ponto de equilíbrio. A partir daí, o protocolo passa a buscar o seu ponto quiescente de forma mais conservadora. O algoritmo que implementa esta etapa do controle de congestionamento chama-se *congestion avoidance*.

O *congestion avoidance* manipula a janela de congestionamento seguindo a regra de *additive increase–multiplicative decrease*. Em situações de congestionamento (sinalizadas por perda de pacotes) a janela é reduzida de forma exponencial. Do contrário, a janela aumenta linearmente.

Isto se justifica pelo seguinte: pode-se demonstrar que quando a rede está congestionada, as filas nos roteadores aumentam exponencialmente. Portanto, a redução na taxa de envio também deve ser exponencial de modo a permitir que as filas se estabilizem. Por outro lado, a rede não indica ao transmissor que ele está sub-utilizando a banda passante disponível, logo deve haver uma maneira de se aumentar a taxa de transmissão, de modo que se possa descobrir o limite atual de banda passante. Um aumento exponencial, por mais lenta que fosse a constante de tempo, levaria a janela a aumentar muito rapidamente, causando instabilidade na rede.

A taxa adotada para o *multiplicative decrease* é de 0,5. Este número foi escolhido de acordo com o seguinte raciocínio: quando se verifica uma situação de congestionamento, o transmissor pode estar iniciando a conexão (em *slow start*), ou estar no estado estacionário. No primeiro caso, em que um pacote é perdido no *slow start*, reduzir a janela à metade significa voltar a janela para o tamanho anterior, que sabemos que não provocou perdas. A partir deste ponto o aumento da janela passa a ser mais lento, buscando o equilíbrio. Na situação de uma conexão em estado estacionário, a perda de um pacote indica que a banda passante disponível para a conexão diminuiu, provavelmente porque outra conexão foi iniciada e está disputando os recursos da rede. Supondo que a primeira conexão estivesse sozinha e que agora divida a banda com uma outra, é correto reduzir a janela à metade pois a banda passante foi reduzida nesta proporção. Este é o pior caso no estado estacionário, já que se houvesse duas conexões e uma terceira fosse iniciada, a redução de banda seria de 1/3, e assim por diante. A redução pela metade é mais conservadora nestes casos, ajudando a garantir a estabilidade da rede.

O incremento do *additive increase* é de $1/cwnd$, para cada ACK recebido. Isto se traduz em um incremento de no máximo 1 pacote por RTT. Como o TCP é orientado a bytes, e não a pacotes, este incremento será de MSS*MSS/$cwnd$, que incrementará a janela de no máximo MSS a cada RTT.

Para compreender este fato devemos inicialmente notar que em uma janela temos n segmentos, sendo $n = cwnd/\mathrm{MSS}$. O valor novo de cwnd será então $cwnd_{nova} = cwnd + A * \mathrm{MSS*MSS}/cwnd$, onde A é o número de ACKs recebidos. Se toda a janela de dados enviada for recebida com sucesso, então o valor de A será o mesmo valor de n, o número de segmentos enviados na janela. Assim:

$$cwnd_{novo} = cwnd + n * \frac{MSS * MSS}{cwnd} = cwnd + \frac{cwnd}{MSS} * \frac{MSS * MSS}{cwnd} = \qquad \text{(III.4)}$$

e mostrou-se então que:

$$cwnd_{novo} = cwnd + MSS$$

Devemos notar que esta é na realidade uma aproximação, uma vez que os ACKs não são recebidos no mesmo instante. Desta forma a janela *cwnd* aumenta gradualmente com o recebimento dos reconhecimentos, com uma taxa de aproximadamente 1 segmento por janela de ACKs recebida.

O algoritmo de *congestion avoidance* então é realizado da seguinte forma:

1. Quando ocorrer um estouro de temporização (*time-out*), *cwnd* recebe *cwnd*/2;
2. Para cada novo ACK recebido (não duplicado), incrementa-se *cwnd* de 1/*cwnd*;
3. A janela de transmissão é a menor entre a janela do receptor (*awnd*) e *cwnd*.

Na prática, o *slow start* e o *congestion avoidance* são implementados em conjunto. A fusão dos dois algoritmos é bastante simples. Além de *cwnd*, que os dois usam, adiciona-se a variável *ssthresh* (*slow start threshold*, o limite do *slow start*), que determina qual dos dois estará ativo.

A implementação em conjunto fica:

1. Quando ocorrer um *time-out*, *ssthresh* recebe *cwnd*/2;
2. Se *cwnd* < *ssthresh*, a janela é manipulada pelo *slow start*;
3. Se *cwnd* ⩾ *ssthresh*, a janela é manipulada pelo *congestion avoidance*.

Assim, o *slow start* abre a janela exponencialmente até que haja um *time-out*. Quando isto ocorre, a janela volta para 1 e aumenta exponencialmente até metade da janela que provocou a perda. A partir daí, o *congestion avoidance* assume o controle, aumentando a janela lentamente de modo a conseguir mais banda passante disponível na rede.

A figura III.4 ilustra o comportamento dos dois mecanismos operando em conjunto [18] [19]. Inicialmente o mecanismo de *slow start* faz com que a *cwnd* cresca de forma exponencial até a ocorrência de uma perda. Nesse momento o TCP nota que a janela

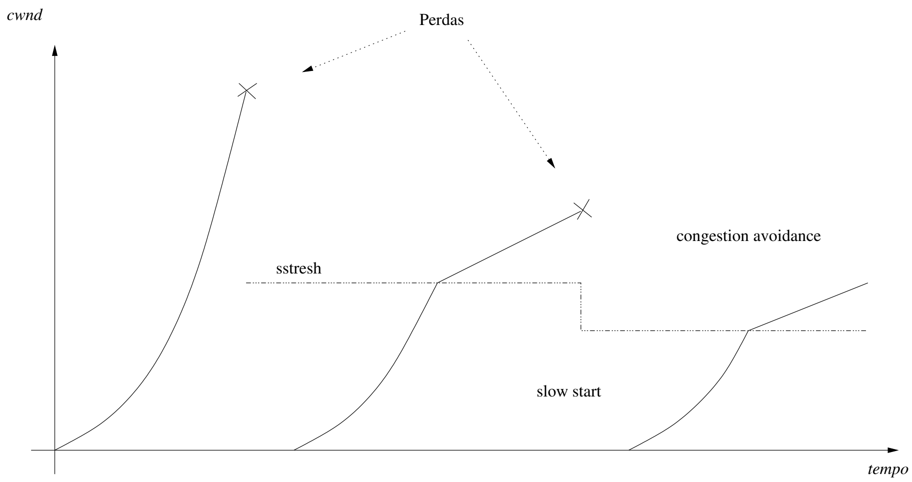


Figura III.4: Mecanismos de *slow-start* e *congestion avoidance* atuando em conjunto.

ultrapassou a capacidade da rede. Assim a variável *ssthresh* recebe metade do valor da janela no instante da detecção da perda. Em seguida a janela é reinicializada em 1 segmento e seu crescimento, até que se atinja o limiar de *sstresh*, se dá de forma exponencial de acordo com o *slow start*. Quando o tamanho de *cwnd* atinge o valor de *sstresh* o protocolo passa a operar em *congestion avoidance*, realizando um aumento linear da janela. Se uma perda for notada já na fase de *congestion avoidance*, o comportamento do TCP será o mesmo.

III.2.4 Fast Retransmit

O mecanismo *fast retransmit* visa a apressar a retransmissão de segmentos quando a rede se encontra em uma situação de congestionamento moderado.

É importante lembrar que um agente TCP deve gerar um ACK (chamado ACK duplicado) ao receber um segmento fora de ordem. Normalmente, ACK(n) tem a conotação de que todos os segmentos até $n - 1$ foram recebidos, e aguarda-se a transmissão de n. Se o transmissor já recebeu ACK(n) e recebe outro, isto significa que um segmento com número maior que n foi recebido, mas n não. Estes ACKs duplicados informam então ao transmissor que um segmento foi recebido fora de ordem.

Como a rede IP é sem conexão, para cada pacote se faz o cálculo da melhor rota e é natural que alguns pacotes cheguem fora da ordem em que foram enviados. Logo, o TCP não sabe se o ACK duplicado foi causado por um pacote perdido ou por uma simples reordenação. Assume-se que no caso de pacotes enviados fora de ordem serão recebidos no máximo dois ACKs duplicados até que o segmento fora de ordem seja processado e um novo ACK seja gerado. Um número de ACKs duplicados maior do que o limite de três é interpretado pelo TCP como um indicativo de perda de pacote, e o segmento é retransmitido. Isto evita que se tenha que esperar o estouro do temporizador, e a retransmissão é feita mais rapidamente.

III.2.5 Fast Recovery

O *fast recovery* foi uma modificação ao algoritmo anterior. Seu objetivo é melhorar a eficiência de utilização do meio, evitando que o canal se esvazie em decorrência de uma perda isolada.

Devemos notar novamente que o controle de congestionamento do TCP usa como indicador de congestionamento na rede uma perda de pacote. A perda de pacote, por sua vez, pode ser detectada de duas formas: por um *time-out* ou pela recepção de um determinado número de ACKs duplicados. Neste último caso, além de indicar a perda de um pacote, os ACKs duplicados carregam outra informação: que pacotes válidos deixaram a rede. Como o controle de congestionamento do TCP é baseado no princípio da conservação de pacotes na rede, o fato de um pacote ter sido retirado implica que se pode transmitir outro pacote pela rede.

Mais que isso, os ACKs duplicados indicam que mesmo que um pacote tenha sido perdido, existe um fluxo de dados entre os dois pontos-finais da conexão. Isto significa que o congestionamento que existe é moderado, e portanto não é necessário realizar novamente o *slow start* para esvaziar o canal.

O *fast recovery* é implementado em conjunto com o *fast retransmit*, da seguinte forma:

1. Ao receber três ACKs duplicados consecutivos, *ssthresh* recebe *cwnd*/2;

2. Retransmite-se o segmento supostamente perdido;

3. *Cwnd* recebe *ssthresh*+3*MSS;

4. A cada novo ACK duplicado, incrementa-se *cwnd* de 1*MSS; se for possível, transmite-se um novo segmento;

5. Ao chegar um novo ACK, *cwnd* recebe *ssthresh*;

Deve-se notar que o primeiro e o último passo do algoritmo acima correspondem ao *congestion avoidance*, ou seja, a janela é reduzida à metade. No entanto, para evitar o *slow start*, o *fast recovery* "infla" a janela de congestionamento, incrementando-a de um MSS para cada ACK duplicado recebido (passos 3 e 4). Deste modo, para cada pacote que deixa a rede, um novo é inserido. No passo 3, *cwnd* é incrementada de 3*MSS de uma vez por causa dos três ACKs duplicados que são necessários para disparar o *fast recovery*, indicando que três pacotes já deixaram a rede. O *fast retransmit* está implementado no passo 2.

III.3 Opção de Reconhecimentos Seletivos

O TCP tradicional realiza um esquema de reconhecimentos cumulativos, em que ACK(n) indica que todos os segmentos até $(n-1)$ foram recebidos com sucesso e o transmissor pode transmitir o segmento (n). Isto impõe uma grave limitação de desempenho em situações onde há mais de uma perda em uma mesma janela de transmissão, porque o transmissor tem que esperar um RTT para identificar cada segmento perdido. Isto ocorre porque caso haja diversas perdas, os reconhecimentos são enviados referenciando o primeiro pacote perdido, não dando maiores informações sobre as demais perdas ao emissor. Assim, o emissor reenvia apenas o pacote referenciado pelos ACKs duplicados, tendo que aguardar por um RTT a chegada do reconhecimento do pacote retransmitido, que estará indicando o próximo pacote perdido. Desta forma apenas um segmento perdido pode ser notado a cada RTT. A alternativa para esta limitação seria retransmitir mais do que o necessário, como por exemplo todos os pacotes após aquele primeiro que foi perdido. Medidas desse tipo, entretanto, além de ineficientes podem acabar agravando uma situação de congestionamento existente na rede.

Uma solução para este problema seria o uso de reconhecimentos seletivos (*selective acknowledgement*, ou SACK) por parte do TCP. Com esta modificação o emissor tem mais informações sobre quais segmentos foram recebidos corretamente e quais não foram, podendo portanto retransmitir somente os pacotes necessários, aumentando a eficiência na recuperação de perdas.

Esta modificação usa o campo OPTIONS (opções) do cabeçalho do TCP para incluir as informações de reconhecimento seletivo. Existem dois tipos de campos opcionais que são usados pelo TCP SACK: um para estabelecer se a opção de SACK será ou não usada (enviada ao iniciar uma conexão), e o SACK propriamente dito, que passa as informações sobre os segmentos reconhecidos. A estrutura deste último campo está ilustrada na figura III.5.

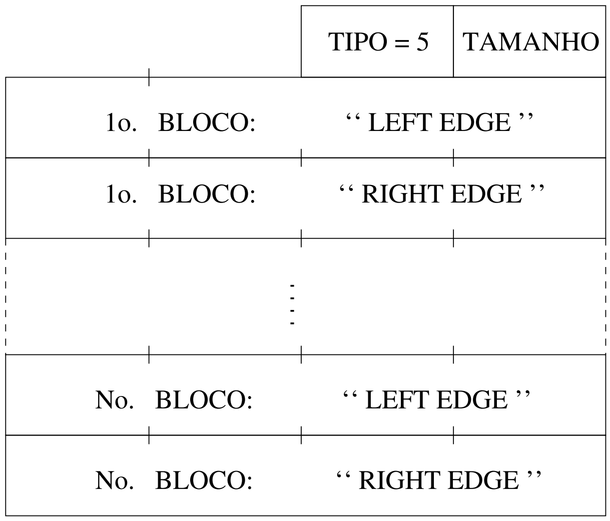


Figura III.5: Campo de reconhecimentos seletivos, o SACK.

Como se vê na figura, o receptor envia informações sobre blocos de dados que foram recebidos e que se encontram isolados. O LEFT EDGE (*left edge of block*, borda esquerda do bloco) é o número de seqüência do primeiro byte deste bloco, enquanto que o RIGHT EDGE (*right edge of block*, borda direita do bloco) é o número de seqüência seguinte ao último byte deste bloco. Isto mostra uma analogia com o reconhecimento normal, que tem o número seguinte ao último recebido. É importante frisar que o campo ACK do cabeçalho mantém o seu significado, ou seja, ele tem o número do primeiro segmento que não foi recebido em seqüência. Quando o receptor preenche as lacunas anteriormente expressas no campo de SACK, ele avança normalmente o número do ACK para denotar um reconhecimento cumulativo.

O campo de opções do TCP pode conter no máximo 40 bytes. O campo de SACK que especifique n blocos terá 8*n + 2 bytes, o que dá um limite teórico de quatro blocos que podem ser reconhecidos. Como na prática outras opções são usadas, o número de blocos, em geral, fica limitado a três.

III.3.1 Comportamento do Receptor

O objetivo da opção de reconhecimento seletivo é que o transmissor tenha mais informações sobre quais segmentos foram recebidos de modo a definir uma estratégia de retransmissão mais eficiente. Portanto, existem regras bem definidas para a geração dos SACKs por parte do receptor, para que o transmissor tenha o retrato mais recente e fiel do estado do buffer do receptor.

Ao gerar um SACK, o receptor deve obedecer às seguintes regras:

1. O primeiro bloco deve conter o segmento que disparou este ACK (a menos que o último segmento tenha dado origem a um ACK normal), ou seja, o primeiro bloco mostra a mudança mais recente no estado do buffer do receptor;

2. O receptor deve incluir o maior número possível de blocos;

3. Os blocos de reconhecimento seletivo devem ser preenchidos do mais recente (primeiro bloco) para o mais antigo; deste modo, um bloco de dados será incluído em no mínimo três ACKs, o que agrega redundância em caso de perda de ACKs.

Com relação à última regra, deve-se notar que mesmo que sejam perdidos todos os ACKs que reportem um bloco de SACK, a pior conseqüência que isso pode ter é a retransmissão desnecessária dos segmentos pertencentes a este bloco. Como sem o SACK, essas retransmissões desnecessárias seriam feitas de qualquer modo, o comportamento do TCP com SACK neste pior caso é igual ao dos TCPs normais.

O receptor mantém um buffer com os segmentos recebidos à espera de que sejam lidos pela aplicação. Quando isso acontece, os dados lidos são apagados do buffer. Em caso de falta de espaço de buffer, o receptor pode descartar segmentos que tenham sido reconhecidos através da opção de SACK. Neste caso, o próximo ACK gerado deve ter no primeiro bloco a mudança mais recente nos segmentos reconhecidos, conforme as regras já citadas.

III.3.2 Comportamento do Transmissor

No TCP tradicional, o transmissor mantém um buffer de retransmissão com os segmentos que foram enviados e ainda não foram reconhecidos. Ao receber um ACK, ele descarta todos os segmentos cujos números de seqüência sejam menores que o número indicado no ACK.

O reconhecimento seletivo estabelece que para cada segmento em seu buffer o transmissor tenha um flag "SACKed", que indica que o segmento foi reportado em um bloco de SACK. Em outras palavras, quando o transmissor recebe um ACK com reconhecimentos seletivos, ele deve colocar em 1 este flag para todos os segmentos que estejam dentro dos blocos especificados na opção de SACK. Deste modo, quando houver retransmissões, somente os segmentos que tiverem o flag SACKed em 0 são passíveis de retransmissão.

Quando houver um *time-out*, o transmissor deve zerar todos os flags "SACKed" da fila de retransmissão, pois isto pode indicar que o receptor descartou os segmentos anteriormente reconhecidos. Neste caso, o transmissor deve retransmitir o segmento indicado no último ACK, independentemente do estado dos flags.

Os segmentos que são reconhecidos seletivamente não podem ser apagados do buffer de retransmissão, porque o receptor pode descartá-los. Um segmento só é apagado quando for reconhecido por um ACK normal.

III.3.3 Exemplos da Utilização do SACK

Para tornar mais claro o entendimento do mecanismo de reconhecimentos seletivos, o SACK, serão mostrados alguns exemplos do seu funcionamento.

Nos casos examinados a seguir sempre será suposto que a borda esquerda da janela tem o número 5000 e que uma rajada de 8 segmentos, cada um com 500 bytes de dados, foi enviada.

Num primeiro exemplo, supõe-se que os 4 primeiros segmentos chegaram com sucesso, enquanto que os 4 últimos foram descartados na rede. Nesta situação o receptor irá gerar um ACK comum, reconhecendo o valor de seqüência 7000, sem utilizar a opção de SACK. Isto ocorre porque o bloco de dados recebido foi contíguo. Do ponto de vista do receptor, os dados que não chegaram simplesmente não foram enviados. Ele não tem dados suficientes para inferir se houve perdas e, portanto, não deve tomar nenhuma providência.

Já em uma outra situação, quando por exemplo apenas o primeiro segmento for perdido, sendo que os 7 seguintes chegam com sucesso, a opção de SACK será utilizada de fato. O receptor irá gerar um pacote de ACK referenciando o número de seqüência 5000 para cada novo pacote que chegar, utilizando entretanto o campo de SACK para indicar quais os blocos recebidos com sucesso. Desta forma os pacotes de ACK enviados irão utilizar o SACK como visto na tabela III.1.

Um exemplo mais interessante ocorre quando tem-se múltiplas perdas na rajada enviada. Supondo-se a perda do 2^o, 4^o, 6^o e 8^o segmentos, teríamos uma situação onde o receptor iria enviar um ACK normal para o primeiro bloco recebido, indicando que espera o recebimento do bloco com número de seqüência 5500. Com o recebimento dos demais

Tabela III.1: Utilização do campo SACK no caso da perda do primeiro segmento enviado.

Segmento Recebido	ACK	Borda Esquerda	Borda Direita
5000	(Perdido)		
5500	5000	5500	6000
6000	5000	5500	6500
6500	5000	5500	7000
7000	5000	5500	7500
7500	5000	5500	8000
8000	5000	5500	8500
8500	5000	5500	9000

pacotes, o receptor iria continuar enviando ACKs com seqüência 5500, indicando porém na opção SACK os pacotes já recebidos com sucesso. A tabela III.2 ilustra a situação.

Tabela III.2: Utilização do campo SACK no caso da perda diversos segmentos.

Segmento	ACK	1^o Bloco		2^o Bloco		3^o Bloco	
		B. Esq.	B. Dir.	B. Esq.	B. Dir	B. Esq.	B. Dir
5000	5500						
5500	(Perdido)						
6000	5500	6000	6500				
6500	(Perdido)						
7000	5500	7000	7500	6000	6500		
7500	(Perdido)						
8000	5500	8000	8500	7000	7500	6000	6500
8500	(Perdido)						

O *buffer* do receptor, ilustrado na figura III.6, apresentaria três blocos de dados recebidos que não seriam reconhecidos caso não se utilizasse a opção de SACK. Na realidade, caso não se empregasse o SACK, seriam gerados ACKs indicando a chegada dos blocos de dados não contíguos, porém os ACKs apresentariam números de seqüência referenciando 5500. Desta forma não seria indicado, neste momento, quais os blocos recebidos com sucesso.

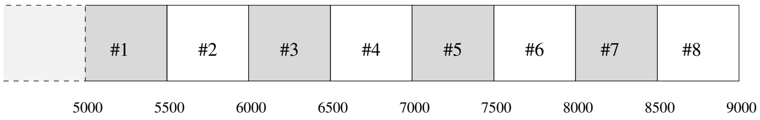


Figura III.6: Representação do estado do *buffer* do receptor.

Se em seguida o 4^o pacote for recebido (o que pode ter ocorrido tanto por causa de entrega fora de ordem pela camada de protocolo inferior, quanto por uma possível perda do 2^o pacote da seqüência, na sua retransmissão, tendo o 4^o segmento sido também retransmitido, porém entregue) o receptor irá compor os pacotes de reconhecimento como exposto na tabela III.3.

Tabela III.3: Blocos reconhecidos pelo receptor com a chegada do 4^o segmento.

Segmento	ACK	1^o Bloco		2^o Bloco		3^o Bloco	
		B. Esq.	B. Dir.	B. Esq.	B. Dir	B. Esq.	B. Dir
6500	5500	6000	7500	8000	8500		

Quando em seguida o 2^o segmento for recebido, o pacote de ACK irá apresentar os dados mostrados na tabela III.4.

Tabela III.4: Pacote de ACK enviado com a chegada do 2^o segmento.

Segmento	ACK	1^o Bloco		2^o Bloco		3^o Bloco	
		B. Esq.	B. Dir.	B. Esq.	B. Dir	B. Esq.	B. Dir
5500	7500	8000	8500				

Este exemplo ilustra bem o comportamento do TCP SACK e suas vantagens em termos de eficiência de retransmissão.

III.4 Implementações do TCP

Com a evolução dos algoritmos de controle de congestionamento descritos anteriormente, diferentes versões do protocolo TCP foram sendo implementadas. De acordo com os novos mecanismos propostos, o *software* do TCP veio sofrendo modificações de modo a incorporar as modificações sugeridas.

Nesta seção serão brevemente descritas as versões mais conhecidas do protocolo, ilustrando sua evolução, no que diz respeito aos algoritmos de controle de congestionamento.

III.4.1 Tahoe

As primeiras implementações do TCP não incluíam qualquer mecanismo de controle de congestionamento. Elas simplesmente usavam a janela do receptor para transmissão e um temporizador de retransmissão. Esta retransmissão era feita de acordo com a estratégia "*go-back-n*", ou seja, todos os segmentos após o segmento dado por perdido eram retransmitidos. Não havia nenhum tipo de busca de realimentação de informações que indicassem a situação da rede. Além disso, os temporizadores do TCP eram deficientes no cálculo da estimativa do RTT, o que provocava problemas nas retransmissões.

O Tahoe foi a primeira implementação do TCP a incluir o controle de congestionamento. O Tahoe usa o *slow start*, o *congestion avoidance* e o *fast retransmit*, além de modificações no estimador do RTT.

III.4.2 Reno

O TCP Reno foi o sucessor do Tahoe, incorporando uma modificação ao *fast retransmit*: o *fast recovery*. O TCP Reno é a versão do protocolo que encontra-se mais difundida na Internet atualmente.

III.4.3 New Reno

O TCP New Reno é uma otimização do TCP Reno, para o caso onde ocorrem múltiplas perdas de pacotes em uma única janela de transmissão. O New Reno inclui uma modificação no algoritmo de *fast recovery* que elimina a necessidade do Reno esperar por um estouro de temporização no caso de múltiplos descartes [20].

O algoritmo do New-Reno utiliza o conceito de reconhecimento parcial. Deve ser notado que quando o TCP percebe que ocorreu a perda de um pacote através do recebimento de um determinado número de reconhecimentos duplicados, a próxima informação nova para o emissor só será dada com a chegada do reconhecimento para aquele pacote retransmitido. No caso de uma única perda, esse reconhecimento irá confirmar o recebimento de todos os pacotes transmitidos antes do início do *fast retransmit*. Se ocorrer a perda de múltiplos pacotes, entretanto, o reconhecimento irá confirmar alguns segmentos, mais precisamente, todos os reconhecimentos até a próxima perda. Esse reconhecimento que não confirma todos os pacotes enviados antes do início do *fast retransmit* é chamado de reconhecimento parcial [6].

Pode-se concluir então que reconhecimentos parciais duplicados, da mesma forma que os reconhecimentos duplicados iniciais, indicam muito provavelmente que um outro pacote dentro da mesma janela foi perdido ao longo do caminho.

O TCP Reno não diferencia os reconhecimentos recebidos durante o *fast recovery*, sejam eles parciais ou não, reconhecendo todos os dados pendentes no início do algoritmo de *fast recovery*. Assim, no TCP Reno, a cada reconhecimento parcial recebido o valor da janela cai pela metade, fazendo o protocolo entrar novamente em *fast recovery* mesmo que as perdas ocorram na mesma janela de transmissão. Este fato acaba por prejudicar o desempenho do Reno, pois sua janela é reduzida repetidamente, qunado uma redução já poderia ser suficiente para contornar a situação de congestionamento.

Já o TCP New Reno, responde aos reconhecimentos parciais. Assim, ao receber reconhecimentos parciais, o TCP se mantém no algoritmo de *fast retransmit*, evitando que ocorram múltiplas reduções no valor da janela de congestionamento.

O algoritmo de *fast recovery* do New Reno utiliza então a variável *recover* que guarda o número de seqüência mais alto já transmitido. O algoritmo é descrito a seguir:

1. O recebimento do terceiro ACK duplicado faz com que *ssth* receba *cwnd*/2. Grava-se o maior número de seqüência na variável *recover*;

2. Retransmite-se o segmento perdido. *Cwnd* recebe *ssthresh*+3*MSS - "inflando artificialmente" a janela de congestionamento pelo número de pacotes que deixaram a rede (três);

3. A cada novo ACK duplicado, incrementa-se *cwnd* de 1*MSS; se for possível, transmite-se um novo segmento;

4. Ao chegar um novo ACK:

   Se o ACK inclui o dado cujo número de seqüência ficou guardado em *recover* e fazer *cwnd* receber*ssthresh*. Sair do *fast recovery*;

   Se o ACK não reconhece todos os dados, ele é um ACK parcial. Retransmitir o primeiro segmento não reconhecido, "desinflar" a janela de transmissão pelo número de reconhecimentos recebidos. Somar 1*MSS à janela de congestionamento e enviar novos segmentos, se o tamanho de *cwnd* permitir. Não sair do *fast recovery* (ou seja, se algum novo ACK duplicado chegar, repetir a partir do passo 3 acima).

A idéia principal do New Reno consiste então em fazer com que, quando múltiplos pacotes são perdidos em uma janela de dados, o fluxo de transmissão se recupere sem que aconteça um esgotamento do temporizador, retransmitindo um pacote por RTT até que todos os dados pendentes tenham sido reconhecidos.

Vale notar que o TCP Reno tenta ser mais agressivo do que o TCP Tahoe. Da mesma forma o TCP New Reno tenta ser mais agressivo do que o Reno e o Tahoe. Porém, como será visto nas simulações, existem diferentes situações que favorecem o desempenho de cada implementação.

Para completar a definição das implementações Reno e New Reno do protocolo TCP, pode ser estabelecido um parâmetro chamado *maxburst* que limita o número máximo de pacotes que podem ser enviados em uma rajada de dados após um recebimento de um reconhecimento, mesmo que o valor da janela permita. Este parâmetro na verdade só é necessário no momento do envio de pacotes da primeira janela de dados após o término de execução do algoritmo de *fast recovery*.

III.4.4 Vegas

Uma implementação proposta para o TCP que apresenta algumas modificações mais profundas nos mecanismos tradicionais deste protocolo é o TCP Vegas, descrito em [21] e [7]. As alterações do Vegas resumem-se ao lado do transmissor, ou seja, sua política de emissão de reconhecimentos é a mesma do TCP Reno. Isto facilitaria sua adoção por parte da comunidade da Internet, já que os seus benefícios podem ser desfrutados mesmo que na outra ponta haja uma versão mais antiga (ao contrário, por exemplo, do SACK). O transmissor segue as linhas gerais propostas por [3]. O Vegas possui um algoritmo de início de conexão, um para o estado estacionário e um para controlar as retransmissões. Entretanto, os três algoritmos são bastante diferentes dos vistos no TCP Reno.

Congestion Avoidance

O mecanismo de *congestion avoidance* do Vegas é baseado no cálculo da estimativa da vazão da conexão. A partir desta estimativa, são arbitrados um limite mínimo e um máximo de vazão por parte da conexão TCP. O Vegas busca manter a vazão dentro destes limites através da manipulação da janela de congestionamento (*cwnd*).

A estimativa das filas dos roteadores é feita a partir de outros dois valores: a vazão esperada e a vazão real da conexão. A vazão esperada é definida como `Expected = WindowSize/BaseRTT`, onde `WindowSize` é o tamanho atual em bytes da janela de congestionamento (supostamente igual ao número de bytes no meio) e `BaseRTT` é o menor valor de RTT medido para uma dada conexão (que representa o RTT da conexão sem que haja congestionamento). Para o cálculo da vazão real, o Vegas utiliza o RTT real. Para efetuar essa estimativa, ele registra o tempo em que um segmento de referência foi enviado e conta os bytes enviados até a chegada do ACK relativo àquele segmento, cujo tempo também é registrado. A vazão real é calculada como o número de bytes enviados sobre o RTT. Este cálculo é efetuado uma vez a cada RTT.

Então, o Vegas compara a vazão real com a esperada, gerando o parâmetro `Diff = Expected - Actual`. Por construção, `Diff` será sempre igual ou maior que zero, pois a vazão real maior que a esperada implica que o último RTT é menor que BaseRTT, e como este último é atualizado sempre, isto não pode ocorrer. São arbitrados também os parâmetros α e β, que serão usados como base para determinar se a janela de congestionamento deve aumentar, diminuir ou permanecer igual. Se `Diff` $< \alpha$, o Vegas aumenta a janela de congestionamento de forma linear. Se `Diff` $> \beta$, a janela de congestionamento é reduzida também de forma linear, ao contrário do *congestion avoidance* tradicional, que segue a regra do additive increase multiplicative decrease. Se $\alpha <$ `Diff` $< \beta$, a janela de congestionamento fica inalterada.

Como são usados para comparação com taxas de vazão, os parâmetros α e β são expressos em KB/s. Entretanto, é mais fácil entender o raciocínio por trás deste algoritmo se pensarmos em termos do espaço de *buffer* ocupado nos nós intermediários. Tomemos como exemplo uma conexão com BaseRTT=100ms e MSS=1KB. Para uma vazão de 10KB/s, podemos dizer que um segmento se encontra em trânsito na rede, ocupando um *buffer* de um roteador. Se α=30KB/s e β=60KB/s, isto significa que a conexão deverá sempre ter entre três e seis segmentos ocupando *buffers* nos roteadores por onde passa a conexão. Se estiver ocupando menos de três, a conexão não está usando suficientemente a largura de banda disponível na rede. Por outro lado, se estiver ocupando mais de seis *buffers*, estará gerando congestionamento.

Na prática, os valores de α e β são de fato definidos em termos de *buffers* ao invés de vazões. Durante a fase chamada de *linear increase/decrease*, os valores são α=1 e β=3. Isto quer dizer que a conexão busca ocupar ao menos um e no máximo três *buffers* na rede. A importância de ter α maior que zero é que, podendo ocupar ao menos um *buffer* na rede, a conexão pode reagir mais rapidamente a um aumento da banda disponível sem ter que esperar um RTT para que ocorra o aumento da janela de congestionamento. O valor de β é três para ocupar um mínimo de *buffers* sem que a janela fique oscilando permanentemente, o que aconteceria caso β fosse somente uma unidade maior que α.

Slow Start

A modificação feita ao mecanismo de *slow start* tem como meta evitar que o aumento exponencial no início da conexão gere congestionamento na rede, como pode ocorrer com o TCP Reno, por exemplo. Deste modo, no início de uma conexão do TCP Vegas ocorre um aumento exponencial, mas a cada dois RTTs. Isto é feito aumentando a janela de um segmento para cada ACK recebido (como nos outros TCPs) em um RTT, e mantendo-a fixa durante o outro RTT. Durante o ciclo em que a janela fica inalterada, é realizada a comparação entre a vazão esperada e a real, descrita anteriormente. Quando a vazão real fica abaixo da esperada pelo equivalente a um *buffer* (um MSS), o Vegas muda para o modo *linear increase/decrease*.

Retransmissões

O Vegas tenta superar as limitações de temporização normalmente associadas ao TCP, devido ao uso de um relógio de baixa resolução. Em geral, como dito anteriormente, as implementações do TCP utilizam um relógio que dá um *tick* a cada 500ms. Esta resolução é muito baixa para determinar time-outs com RTTs da ordem de 100ms, que é o que se encontra normalmente [21]. É dado um exemplo, em [21], de testes em que o temporizador deveria ter estourado com 300ms, mas na prática o Reno só percebia o *time-out* em média 1100ms após a transmissão dos segmentos. Em conseqüência disso, o Vegas tenta eliminar a dependência do TCP de temporizadores de baixa precisão.

A estimação do RTT é feita da seguinte maneira: a cada segmento enviado e a cada ACK recebido, o Vegas lê o relógio do sistema e registra o tempo lido. O cálculo do RTT é feito usando os mesmos algoritmos do Reno, mas as medidas de tempo usadas possuem maior exatidão. Isto deve permitir que o TCP trabalhe com uma estimativa de RTT mais exata, o que possibilita as alterações propostas para os algoritmos de retransmissão.

O mecanismo de retransmissão do TCP Vegas é disparado nas situações descritas a seguir:

- Quando é recebido um ACK duplicado, o Vegas confere se a diferença entre a hora atual e a hora registrada para o último segmento não reconhecido é maior que o valor de *time-out*. Se for, o segmento é retransmitido sem que se tenha de esperar por três ACKs duplicados. Este mecanismo é particularmente vantajoso em situações de congestionamento intenso em que haja perdas pesadas de ACKs.
- Quando um ACK normal é recebido, sendo este o primeiro ou segundo após uma retransmissão, o Vegas procede da mesma forma, conferindo a hora de envio do último segmento contra a hora atual. Se a diferença for maior que o valor de *time-out*, o segmento é retransmitido.

O Vegas mantém o mecanismo de time-out do Reno como um último recurso caso os seus mecanismos não percebam a perda de um segmento. No entanto, o objetivo é justamente reduzir ao máximo o número de *time-outs* tradicionais, já que estes normalmente são um mecanismo ineficiente para a determinação das retransmissões.

III.4.5 TCP SACK

O TCP com reconhecimento seletivo não incorpora qualquer mudança nos algoritmos de controle de congestionamento. Neste aspecto, ele é idêntico ao Reno, apenas apresentando a diferença na política de retransmissões decorrente da opção de reconhecimentos seletivos.

Como a proposta do TCP SACK é bastante recente, há diversas implementações em uso corrente. Estas possuem pequenas variações, porém nunca em aspectos essenciais. A robustez e eficiência desta proposta motivaram algumas plataformas a optar por sua utilização como padrão. Um exemplo é o sistema Microsoft Windows 98, que é um dos mais difundidos atualmente. De forma similar, versões experimentais de sistemas como Linux, FreeBSD e Sun Solaris entre outros, estão implementando a opção de reconhecimentos seletivos.

A tabela III.5 apresenta de forma condensada as diferentes implementações do TCP explicadas aqui, para que se possa observar as diferenças entre elas, com relação aos mecanismos de controle de congestionamento e à política de reconhecimentos empregada.

Tabela III.5: Comparação das diferentes implementações do TCP

Implementação	*Slow start*	*Congestion avoidance*	*Fast retransmit*	*Fast recovery*	Política de reconhecimentos
Tahoe	sim	sim	sim	não	cumulativo
Reno	sim	sim	sim	sim	cumulativo
New Reno	sim	sim	sim	sim	cumulativo, diferencia ACKs parciais
Vegas	próprio, mais suave	próprio, *linear increase/ decrease*	não se aplica	não se aplica	cumulativo
SACK	sim	sim	sim	sim	cumulativo com seletividade

As diversas implementações do TCP implicam em diferentes comportamentos nas situações de congestionamento de rede, levando a variações no desempenho do protocolo. Cada implementação possui vantagens e desvantagens sob determinados cenários de congestionamento. O capítulo seguinte apresenta simulações de algumas das variações dos protocolos aqui apresentadas, de modo que se possa comparar o comportamento e observar-se os pontos fortes e deficiências de cada uma delas.

Capítulo IV

Simulações

Este capítulo apresenta uma série de simulações, cujo foco é o estudo do comportamento das diferentes implementações do protocolo TCP em cenários variados de congestionamento. O objetivo deste capítulo é então apresentar um estudo qualitativo do comportamento do controle de congestionamento do TCP em cenários que permitissem diferentes padrões de perdas indicando diferentes níveis de congestionamento da rede.

Foram escolhidos para simulação as implementações Tahoe, Reno e SACK do TCP. O primeiro foi pioneiro na utilização de mecanismos de controle de congestionamento. O Reno, além de incluir o *fast recovery*, é a implementação mais comum, de uso corrente na Internet. O TCP SACK, por sua vez, utiliza um mecanismo extra, o reconhecimento seletivo, que representa uma nova tendência.

A observação do comportamento das implementações é feita no nível de pacotes, tendo como objetivo verificar e ilustrar os mecanismos descritos anteriormente. Mais além, faz-se uma análise em mais alto nível das implicações desses mecanismos no desempenho de uma conexão TCP. Apesar desta análise apresentar caráter fortemente comparativo, procurou-se manter um grau de independência no estudo de cada cenário, de modo a não desprezar detalhes e manter a clareza.

Inicialmente é realizada uma breve apresentação sobre a ferramenta de simulação utilizada. Após isto, descreve-se a topologia empregada nas simulações e em seguida realiza-se a análise das mesmas. No final do capítulo encontra-se mais uma simulação, com a comparação entre a eficiência de diferentes implementações do TCP na transmissão de um arquivo.

IV.1 Simulador de Redes

Para a realização da análise dos mecanismos de controle de congestionamento do TCP foram realizadas simulações utilizando o *software NS* (*Network Simulator*) [9]. Esta ferramenta de simulação de redes foi desenvolvida na Universidade da Califórnia Berkeley, em parceria com o projeto VINT (*Virtual InterNetwork Testbed*), incluindo como colaboradores o USC/ISI, a Xerox PARC e o LBNL. O simulador *ns* foi escolhido para as simulações devido a sua ampla aceitação entre o meio científico. A utilização do simulador é exemplificada no apêndice A.1, juntamente com algumas características importantes de sua implementação.

IV.2 Topologia

O cenário de simulações utilizado é mostrado na figura IV.1. Com esta topologia obtém-se um cenário típico de rede com um gargalo, onde ocorre congestionamento com sobrecarga das filas no nó de entrada do enlace mais lento, causando eventuais descartes de pacotes.

A configuração utilizada apresenta 8 fontes de tráfego TCP, todas com uma mesma versão do protocolo, conectados à um enlace mais lento e com um maior atraso, simulando um enlace de longa distância. Desta forma procura-se retratar um caso típico de duas redes locais interconectadas por um enlace de gargalo. Todas as filas utilizadas nos nós foram do tipo FIFO (*First In First Out*). A vazão do enlace de gargalo é de 155 Mbps com atraso de 30 ms. Os enlaces restantes são de 100 Mbps com 1 ms de atraso.

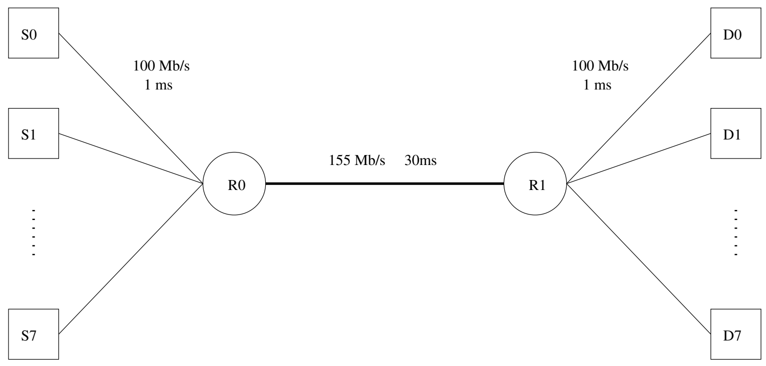


Figura IV.1: Topologia Simulada

IV.3 Congestionamento com Uma Perda

Para a obtenção do cenário de congestionamento com uma perda, alguns parâmetros dos agentes TCP, assim como parâmetros da rede foram ajustados, sendo descritos a seguir:

- Tamanho da fila do roteador: 13 pacotes;
- Tamanho da janela anunciada pelo receptor, a *awnd*: 500 pacotes;
- Tamanho dos pacotes: 500 bytes.

Em todas as simulações realizadas o janela anunciada pelo receptor, a *awin*, foi mantida em 500 pacotes. Este valor foi escolhido alto, para que o controle de fluxo não influenciasse na política de envio dos pacotes, ficando este comportamento exclusivamente guiado pelos mecanismos de controle de congestionamento.

Para os parâmetros descritos acima obteve-se o caso onde apenas um pacote, o de número 29, é descartado da fila do roteador R0.

IV.3.1 Tahoe

A primeira simulação efetuada foi utilizando a implementação do TCP Tahoe, sendo esta apresentada na figura IV.2. Esta figura, assim como as demais analisadas neste capítulo, apresenta em seu eixo horizontal o tempo (em segundos) e em seu eixo vertical a numeração dos pacotes. Desta forma, observando-se o gráfico da figura IV.2 (ou da figura IV.3 que representa um detalhamento desta), descobre-se que no tempo t=0,25 s os pacotes com numeração de 15 a 30 foram enviados. Da mesma forma, nota-se que em t=0,25 s ocorreu o descarte do pacote 29. Os gráficos mostram os pacotes enviados (representados por pontos), os reconhecimentos recebidos (os ACKs, representados por losangos), os pacotes descartados pela rede (indicados com uma cruz), o tamanho da janela de congestionamento em pacotes (*cwnd*, dado pela linha tracejada) e o valor da variável *ssth* (indicado pela linha tracejada marcada com quadrados). As representações citadas são também apresentadas nas legendas das figuras.

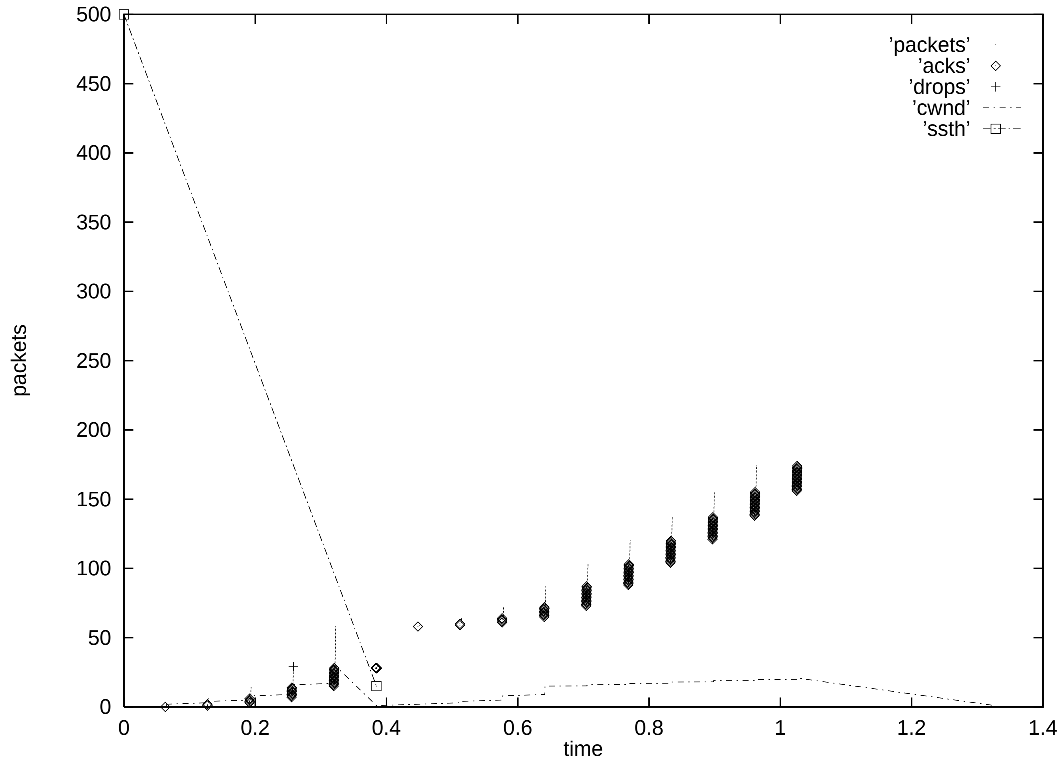


Figura IV.2: Uma perda com a utilização do TCP Tahoe.

No gráfico da figura IV.2 observa-se claramente o crescimento exponencial da janela de congestionamento, a *cwnd*, desde o início da transmissão até o ponto onde ocorre a perda do pacote 29 em t=0,25 s.

Até este descarte, o algoritmo de *slow-start* é executado, onde o valor da janela de

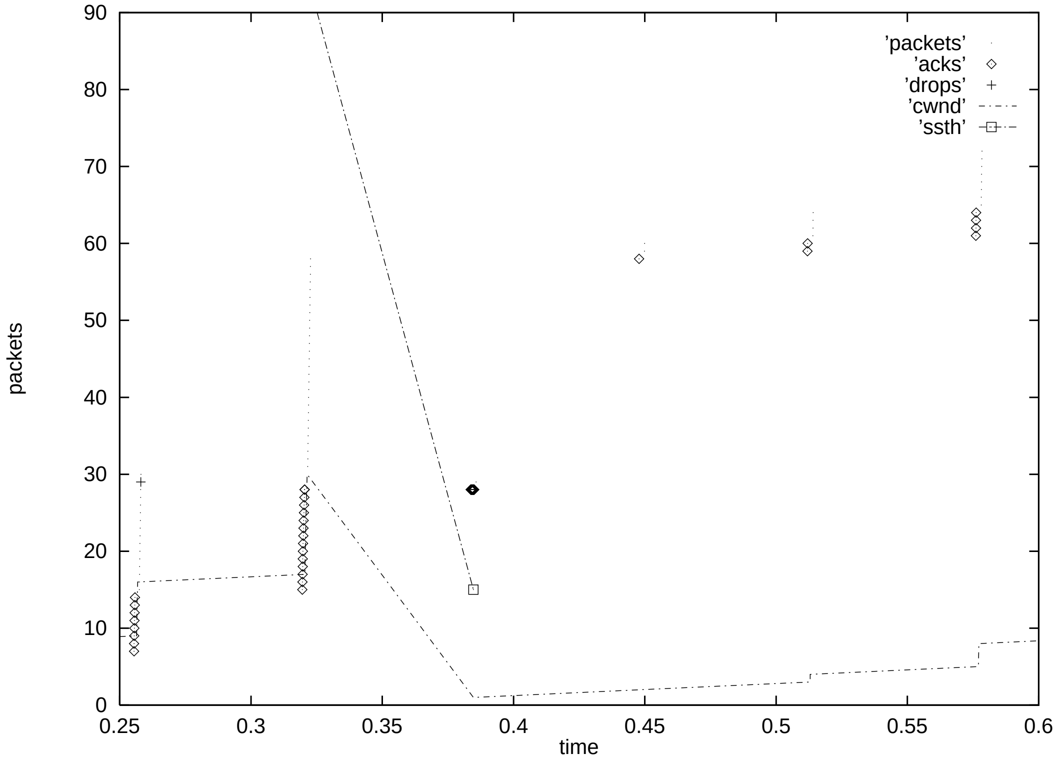


Figura IV.3: Detalhe do comportamento do TCP Tahoe no descarte de um pacote.

congestionamento cresce em potências de 2. Quando a *cwnd* atinge o tamanho de 16 pacotes, ocorre o descarte do segmento de número 29. Desta forma, o receptor passa a enviar ACKs com o reconhecimento do pacote 28 [1] para cada novo pacote que é recebido a partir deste momento. Ao receber o quarto ACK com número 28 (o terceiro duplicado), o emissor infere que ocorreu a perda do pacote 29. Isto só ocorre após o envio da janela inteira, até o pacote 58, quando o tamanho da mesma está em 30 pacotes. Assim o *ssthresh* recebe o valor de *cwnd*/2, passando ao tamanho de 15 pacotes e o pacote 29 é retransmitido. Esta é a ação do algoritmo de *fast retransmit*, evitando que seja necessária a ocorrência de um estouro do temporizador para que a perda do pacote fosse detectada. Neste instante a *cwnd* volta à 1 e o algoritmo de *slow-start* é executado novamente, como apresentado de forma mais detalhada na figura IV.3. Assim a janela de congestionamento aumenta de forma exponencial, porém ao alcancar o valor de *ssthresh* o algoritmo de *congestion avoidance* é iniciado, fazendo com que a janela aumente de forma linear a uma

[1] Deve ser lembrado que os ACKs no *NS* indicam o último pacote recebido e não o próximo à receber.

taxa aproximada de 1 (um) segmento por RTT, como pode ser visto na figura IV.2.

IV.3.2 Reno

A simulação para o TCP Reno é apresentada na figura IV.4, revelando que o protocolo se comporta inicialmente de forma semelhante ao TCP Tahoe, buscando a ocupação do canal com o crescimento exponencial da janela de transmissão. Novamente o emissor nota que ocorreu a perda ao receber o terceiro ACK duplicado para o pacote 28. As diferenças entre as implementações tornam-se visíveis a partir deste ponto. Novamente a variável *ssthresh* assume metade de *cwnd* (15 pacotes) porém, ao contrário do TCP Tahoe, a janela de congestionamento não retorna para 1 pacote e o algoritmo de *slow-start* não é novamente executado. Em contrapartida o protocolo entra em regime de *fast-recovery*, procurando manter a ocupação do canal, uma vez que o congestionamento é apenas moderado. Ao fim do recebimento dos ACKs dos pacotes enviados na janela de transmissão onde ocorreu a perda, as variáveis estão com os valores: *awnd*= 500, *ndup*= 28 e *cwnd*= 15, onde *ndup* é o número de ACKs duplicados recebidos. A janela de transmissão "infla" até o valor min(*awnd, cwnd+ndup*) que é 43. Como já existem 28 segmentos aguardando reconhecimento, o protocolo pode enviar mais 15 novos pacotes, incluindo a retransmissão.

Ao receber o reconhecimento total, do pacote 60, o protocolo passa a operar com o algoritmo de *congestion avoidance* aumentando a *cwnd* (neste momento em 15 pacotes) de forma linear.

Pode ser notado um melhor desempenho na implementação do TCP Reno em relação ao Tahoe. Isto ocorre porque o TCP Tahoe é uma implementação cautelosa em termos de contenção de congestionamento. O Tahoe age de forma a garantir, com larga margem de confiança, a estabilidade da rede mesmo que para isso tenha que prejudicar, em alguns casos, seu desempenho.

Este fato fica claro quando observa-se que para qualquer perda de pacote o algoritmo *slow-start* é iniciado, reduzindo a janela de congestionamento ao tamanho unitário,

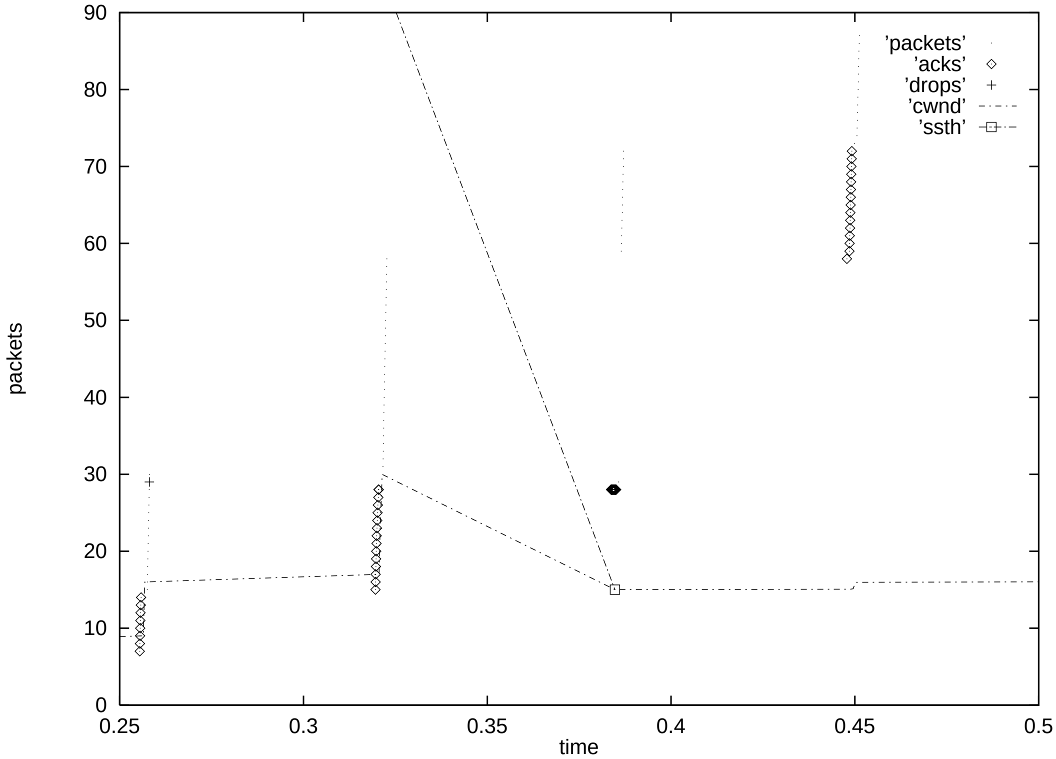


Figura IV.4: Uma perda com a utilização do TCP Reno.

independentemente do nível de congestionamento da rede. Por outro lado o TCP Reno busca manter a ocupação do canal através do mecanismo de *fast recovery*, permitindo uma recuperação mais rápida.

IV.3.3 SACK

Observa-se que não há diferenças entre os comportamentos do TCP Reno e do TCP SACK para o cenário simulado. A figura IV.5 ilustra este fato. Isto ocorre porque neste caso, com o descarte de apenas um pacote, a opção de reconhecimento seletivo não é de fato utilizada.

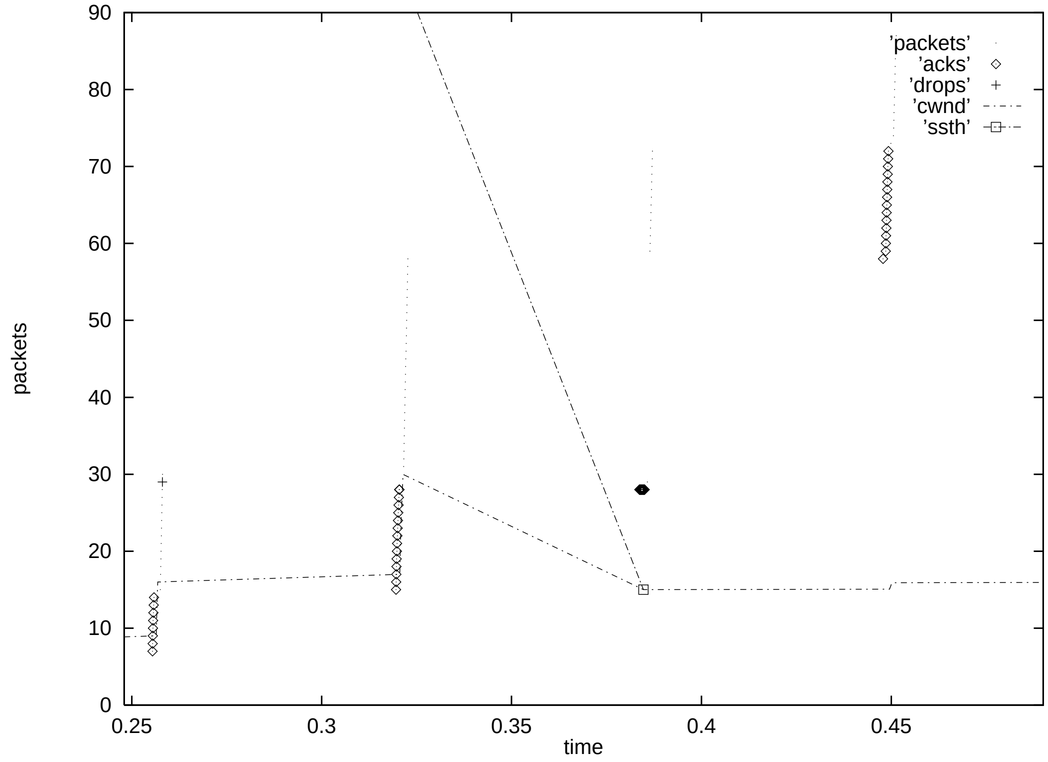


Figura IV.5: Uma perda com a utilização do TCP SACK.

IV.4 Congestionamento com Duas Perdas

O cenário de simulação para congestionamento com a perda de dois pacotes utilizou os parâmetros seguintes:

- Fila do nó R0 para R1: 14 pacotes;
- Tamanho da *awnd*: 500 pacotes;
- Tamanho de cada pacote: 1001 bytes.

Este cenário proporcionou o descarte dos pacotes 28 e 30 no nó de entrada do enlace de gargalo.

IV.4.1 Tahoe

Na figura IV.6 vemos o resultado da simulação deste cenário de perdas para o TCP Tahoe. De forma similar ao que ocorre no caso de uma perda, o algoritmo de *slow-start* é realizado inicialmente, aumentando de forma exponencial a janela de congestionamento até a ocorrência das perdas, que ocorrem na rajada que vai do pacote 15 ao 30. Ao receber o terceiro ACK duplicado para o pacote 27, que só ocorre em t=0,38 s, com o recebimento do ACK disparado por um pacote da próxima janela de transmissão (pacote 33), o algoritmo efetua a retransmissão do pacote 28, reduz o *ssthresh* para metade de *cwnd* (14 pacotes, pois foram recebidos 14 pacotes dos 16 transmitidos, sendo que o reconhecimento do pacote 29 é recebido como ACK duplicado. Como no *slow start* a janela só é incrementada para ACKs novos, teremos nesse instante *cwnd*= 16 + 13= 29.) e reduz a janela de transmissão para um pacote. A partir desse instante o algoritmo de *slow-start* é executado novamente. O reconhecimento recebido pela retransmissão do pacote 28, em t=0,45 s, vai indicar cumulativamente o recebimento do pacote 29 (já transmitido anteriormente). Os outros pacotes já recebidos anteriormente ainda não foram reconhecidos nesse instante, uma vez que o pacote 30 também foi descartado. A janela então aumenta para 2 e os pacotes 30 e 31 são retransmitidos, sendo que o último desnecessariamente, uma vez que já se encontrava armazenado no receptor. É interessante observar que a ausência de informações sobre o estado do *buffer* do receptor não permitiu uma otimização nas retransmissões, forçando o reenvio de alguns pacotes desnecessariamente.

O próximo ACK, em t=0,51 s finalmente reconhece todos os pacotes pendentes e o algoritmo de *slow-start* continua. Quando a *cwnd* alcança o *ssthresh* o algoritmo de *congestion avoidance* é iniciado, fazendo com que a janela aumente de forma linear a uma taxa aproximada de 1 (um) segmento por RTT.

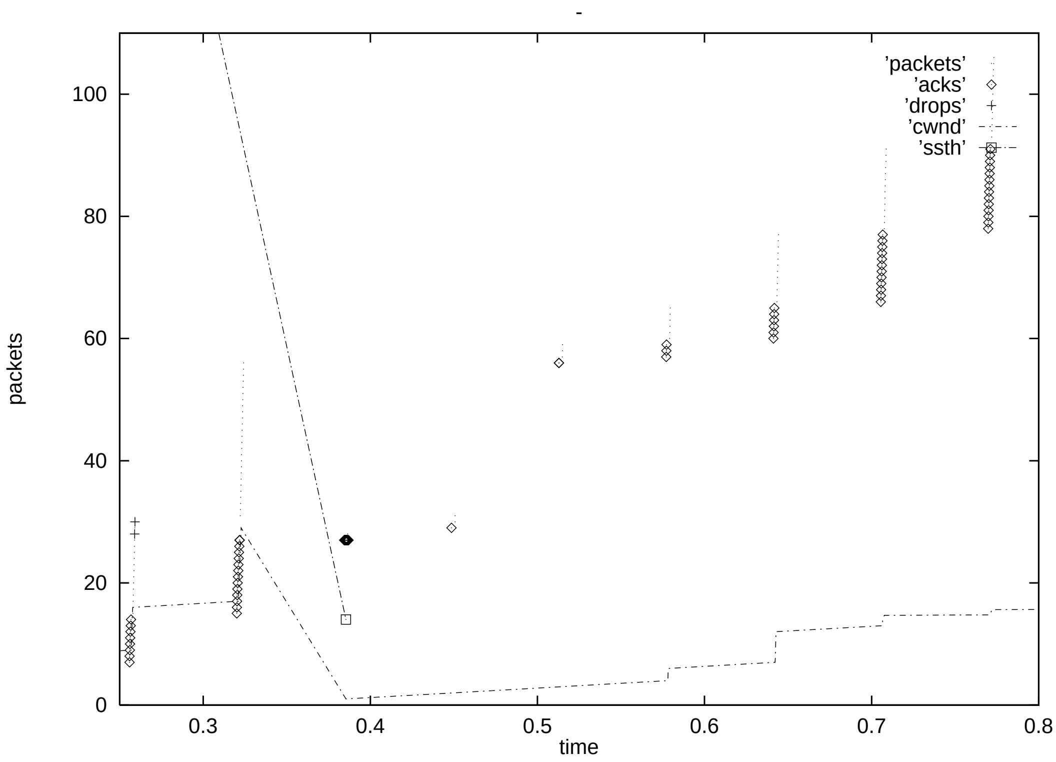


Figura IV.6: Duas perdas com a utilização do TCP Tahoe.

IV.4.2 Reno

A fase inicial do TCP Reno ocorre de forma semelhante à do Tahoe, onde o algoritmo de *slow-start* atua. Quando ocorre a perda e os 3 ACKs duplicados são recebidos é que se iniciam as diferenças de comportamento das implementações. As simulações para o TCP Reno encontram-se ilustradas no gráfico da figura IV.7.

Como citado anteriormente, as perdas ocorrem na rajada que vai do pacote 15 ao 30. O receptor envia os ACKs duplicados para o pacote 27, indicando a perda do pacote 28. Ao receber o terceiro ACK duplicado, em t=0,38 s, o emissor retransmite o pacote 28 e reduz a *cwnd* pela metade. Quando isto ocorre o protocolo já efetuou a transmissão em avanço até o pacote 56, utilizando uma janela de 29 pacotes (do 28 ao 56). Nesse momento o TCP entra em *fast recovery*. Assim serão obtidos *cwnd*= 14, *ndup*= 27 (27 ACKs, pois, 29 pacotes foram enviados mas 2 foram descartados) e *awnd*= 500. Desta forma, min(*awnd*, *cwnd* + *ndup*)= 41. O número de pacotes pendentes, já enviados e aguardando reconhecimento, é entretanto de 29. Assim em t=0,38 s são transmitidos mais 12 (41 - 29) pacotes, excluindo-se a retransmissão do pacote 28.

Os reconhecimentos para os pacotes enviados na janela inflada (de 12 pacotes), entretanto, irão referenciar o pacote 29, que também foi descartado. O recebimento dos ACKs duplicados, em t=0,45 s, permite que o emissor perceba a perda do pacote 30 e assim, mais uma vez, a *cwnd* é reduzida à metade (7 pacotes). As variáveis estão com os valores *cwnd*= 7, *ndup*= 12 e *awnd*= 500. Neste momento tem-se min(*awnd*, *cwnd* + *ndup*)= 19. Existem porém 39 pacotes pendentes. Desta forma, apenas a retransmissão pelo *fast retransmit* pode ser efetuada. Os próximos pacotes só poderão ser enviados após o recebimento do ACK para todos os pacotes pendentes, que ocorre em t=0,52 s. A partir desse instante o protocolo passa para a fase de *congestion avoidance*, aumentando linearmente a janela de transmissão.

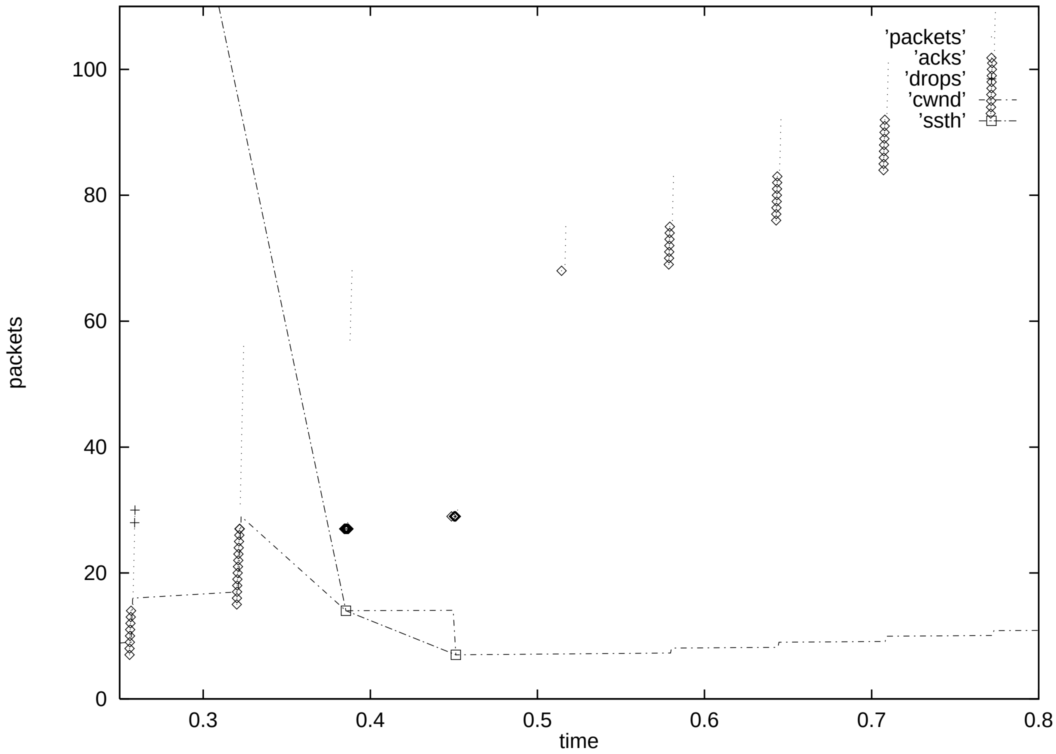


Figura IV.7: Comportamento do TCP Reno no caso da ocorrência de duas perdas.

Observa-se neste caso que a janela de transmissão foi reduzida pela metade duas vezes seguidas, o que não é desejável, uma vez que uma redução já contornaria a situação de congestionamento, não prejudicando demasiadamente a eficiência do protocolo.

Ao se analisar do motivo dessa deficiência do TCP Reno, deve-se observar que o mesmo sai do algoritmo de *fast recovery* ao receber um reconhecimento, sendo este parcial ou não [2]. O TCP Reno não considera o caso do recebimento de reconhecimentos parciais, que são enviados quando ocorrem múltiplas perdas numa mesma janela. Este fato faz com que o protocolo fique saindo e entrando em *fast retransmit/fast recovery* quantas vezes forem as perdas dentro de uma janela de dados, diminuindo cada vez mais a janela de transmissão e prejudicando sua eficiência.

IV.4.3 SACK

O gráfico da figura IV.8 mostra os resultados da simulação para o TCP SACK, no caso do descarte de dois pacotes.

O protocolo se inicia com o algoritmo de *slow-start*, de forma semelhante às outras implementações do TCP, tentando ocupar o mais rapidamente possível o meio de transmissão. Ao receber os ACKs duplicados para o pacote 27, são tomadas as providências relativas ao reenvio de dados, uma vez que a perda do pacote 28 é inferida.

É interessante notar que neste caso a opção por reconhecimentos seletivos é utilizada de fato. Ao receber ACKs duplicados referentes à chegada de novos pacotes, o receptor informa a ausência tanto do pacote 28 como também do pacote 30, confirmando entretanto o recebimento dos demais. Ao receber o terceiro ACK duplicado para o pacote 27, em t=0,38 s, o protocolo retransmite o pacote 28 e reduz a *cwnd* pela metade. Quando isto ocorre o protocolo efetuou a transmissão até o pacote 56, utilizando uma janela de 29 pacotes. Assim tem-se *cwnd*= 14, *ndup*=3 e *awnd*=500. Desta forma min(*awnd*, *cwnd* + *ndup*)=17. O número de pacotes pendentes é entretanto de 29. Assim não podem ocorrer novas retransmissões até a chegada de mais 12 ACKs duplicados. Com o recebimento do 13^o ACK duplicado, a janela é inflada o suficiente para ocorrer a retransmissão. Quan-

[2] Um reconhecimento parcial é aquele que não reconhece todos os dados pendentes, mas apenas aqueles que chegam antes de uma nova perda, como foi explicado na seção III.4.3.

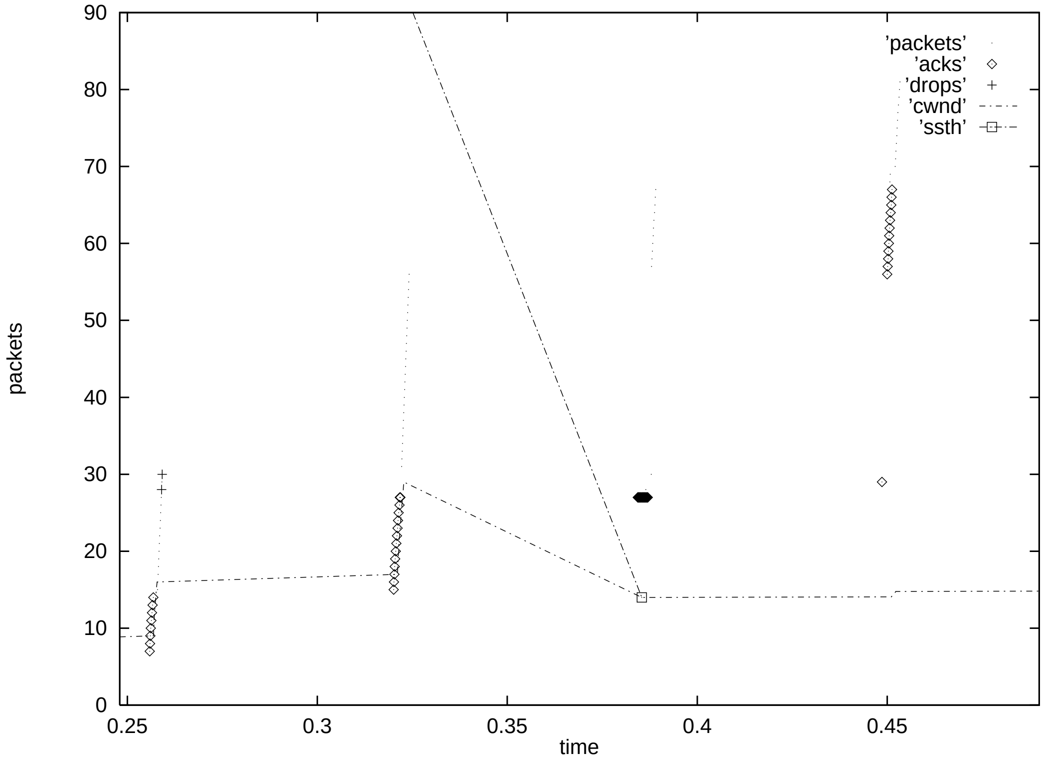


Figura IV.8: Duas perdas com a utilização do TCP SACK.

do isto ocorre, o protocolo automaticamente retransmite o pacote 30, cuja perda havia sido notificada através da opção de reconhecimento seletivo. Com a chegada dos ACKs restantes, a janela é inflada até 41 pacotes (14 + 27), porém como já existem 29 pacotes pendentes, são transmitidos mais 12, sem contar a retransmissão do pacote 28, efetuada pelo *fast retransmit*.

Quando o reconhecimento total para o pacote 56 é recebido o protocolo entra em *congestion avoidance*. As variáveis *ssth* e *cwnd* já haviam recebido metade do valor da janela de transmissão no momento da perda (14 pacotes neste caso, como descrito acima).

IV.5 Congestionamento com Três Perdas

Para a obtenção de um cenário de congestionamento com a perda de três pacotes foram utilizados os parâmetros abaixo. A variação do tamanho da fila no nó de entrada do enlace de gargalo, em relação à simulação anterior, permitiu a obtenção do padrão de perdas desejado.

- Fila do nó R0 para R1: 30 pacotes;

- Tamanho da *awnd*: 500 pacotes;

- Tamanho de cada pacote: 1001 bytes.

Foram descartados os pacotes número 55, 57 e 59.

IV.5.1 Tahoe

O resultado desta simulação pode ser visualizado na figura IV.9. Os três pacotes perdidos foram enviados dentro da mesma janela de transmissão. O transmissor estava na fase de *slow start* e *cwnd*=32. A janela citada incluía os pacotes de 31 a 62.

Em t=0,38 s começam a chegar os reconhecimentos relativos à janela anterior, aumentando a janela de congestionamento da forma esperada. Conforme os ACKs são recebidos, vai sendo enviada a nova janela de transmissão. Como o segmento 55 foi perdido, após o reconhecimento do 54 chegam mais 5 ACKs relativos a este segmento (em resposta aos pacotes 56, 58, 60, 61 e 62). O terceiro ACK duplicado dispara o *fast retransmit*. Neste instante foram enviados os segmentos até 110 e a janela de congestionamento tem tamanho 56. Ao receber o terceiro ACK duplicado, o TCP Tahoe atribui à variável *ssthresh* o valor de *cwnd*/2 (28), leva *cwnd* a 1 e o segmento 55 é retransmitido.

Em t=0,45 s, é recebida uma rajada de 48 ACKs duplicados, indicando o recebimento fora de ordem dos pacotes de 63 a 110. O TCP Tahoe não usa essa informação para fins de controle de congestionamento. O que é relevante para o comportamento do transmissor é a chegada do ACK relativo ao segmento 56. Este reconhecimento indica a recepção do segmento 55, que havia sido retransmitido. O ACK evita o estouro do temporizador e aumenta a janela de congestionamento em um pacote (note que o transmissor está novamente na fase *slow start*).

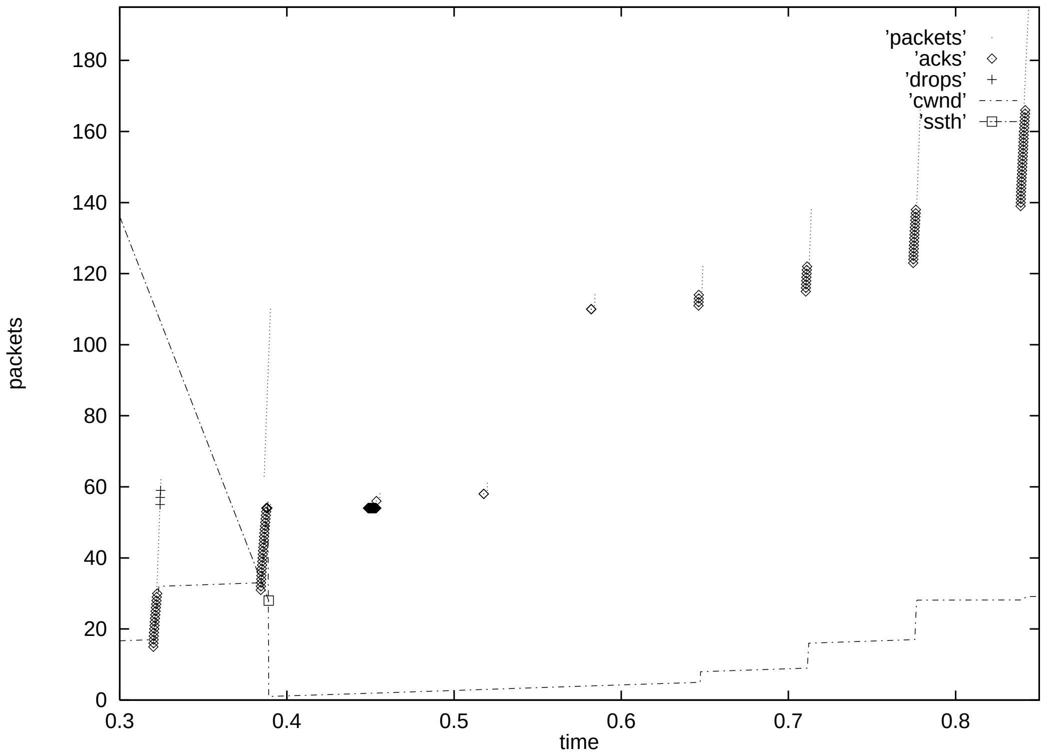


Figura IV.9: Três perdas com a utilização do TCP Tahoe.

Como agora *cwnd*=2, o transmissor envia os dois pacotes seguintes que não foram reconhecidos (57 e 58). Note que o transmissor não faz uso de qualquer informação sobre os pacotes já recebidos, e reenvia o segmento 58 desnecessariamente. Passado mais um RTT, em t=0,51 s, chegam dois ACKs para o segmento 58, um gerado pelo recebimento do 57 e o outro pelo 58. Como somente um desses ACKs indica novos dados, o transmissor abre a janela de mais um pacote, e desta vez envia os segmentos 59, 60 e 61 (esses dois últimos já recebidos anteriormente no outro ponto final). Em t=0,58 s, são recebidos três ACKs relativos ao segmento 110, pois todos os demais pacotes estavam armazenados no buffer do receptor. Como os três ACKs são para o mesmo pacote, a janela é incrementada em um e são enviados quatro segmentos (111 a 114). Assim, o transmissor segue na fase de *slow start* até t=0,84 s, quando *cwnd* chega ao valor de *ssthresh*. Daí ele passa à fase de *congestion avoidance*.

IV.5.2 Reno

O gráfico da figura IV.10 ilustra a simulação do TCP Reno com três perdas em uma janela.

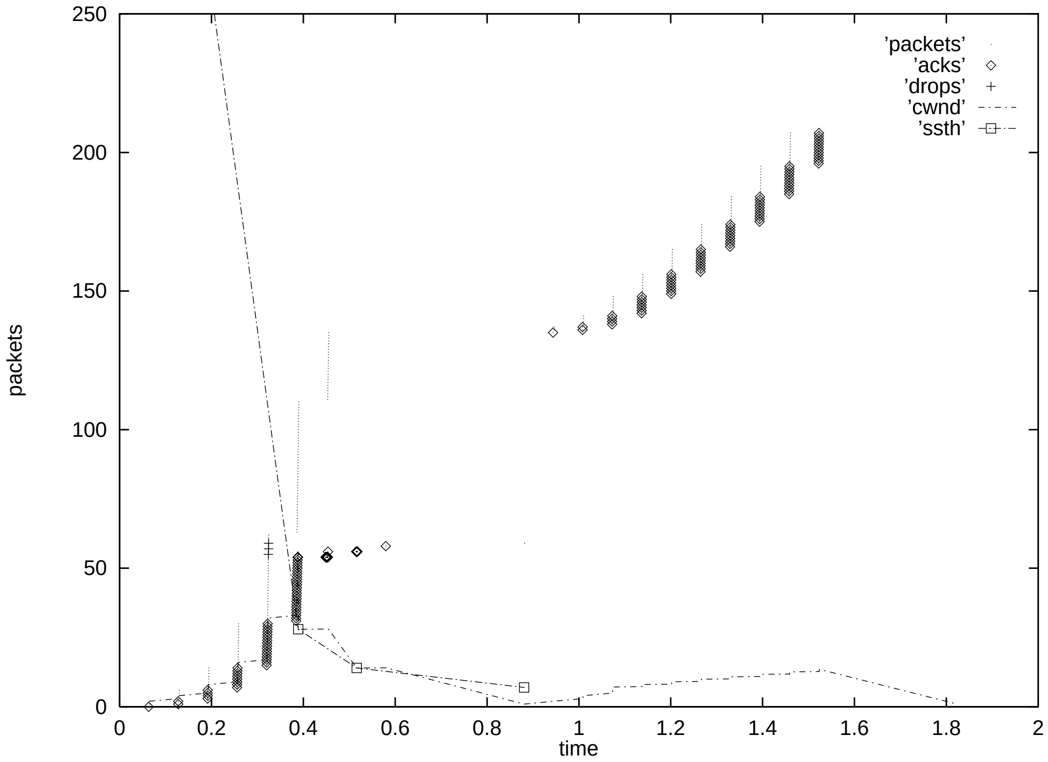


Figura IV.10: Três perdas com a utilização do TCP Reno.

Em t=0,39 s, o transmissor recebe os reconhecimentos correspondentes à janela transmitida em t=0,32 s. Como foram perdidos os pacotes 55, 57 e 59, após o ACK do 54 chegam 5 ACKs duplicados para o mesmo segmento devido ao recebimento de 56, 58, 60, 61 e 62. O terceiro ACK duplicado dispara o *fast retransmit*, levando à retransmissão do pacote 55. Além disso, o Reno entra na fase de *fast recovery*: *ssthresh* recebe *cwnd*/2 (no caso, 28); o novo valor de *cwnd* recebe o valor de *ssthresh* e a janela efetiva de transmissão fica sendo *cwnd*+*ndup*. No caso visto, *ndup*=5, portanto a janela de transmissão é de 33 pacotes.

Deve-se notar que em t=0,39 s os pacotes de 55 até 110 foram transmitidos e não foram reconhecidos, portanto do ponto de vista do transmissor estes pacotes estão ainda no meio de transmissão. Portanto, temos 56 pacotes pendentes. Como a janela atual é 33, será necessário o recebimento de mais 23 ACKs duplicados para que a janela "infle" o suficiente para que o transmissor possa enviar novos dados.

Em t=0,45 s são recebidos 48 ACKs duplicados pelo recebimento dos pacotes de 63 a 110. Portanto, a janela do transmissor é incrementada a 33+48=81. Como havia 56 segmentos pendentes, são enviados 81-56=25 segmentos, cuja numeração vai de 111 a 135. Imediatamente após a chegada destes ACKs duplicados é recebido o ACK relativo ao segmento 56, indicando que a retransmissão do 55 teve sucesso. Este é na realidade um ACK parcial. Porém, como dito anteriormente, o TCP Reno não o distingue de um ACK comum. O transmissor reage a este ACK saindo da fase de *fast recovery*. Neste ponto a janela de transmissão volta a ser igual a *cwnd*, que vale 28. Como *ssthresh* é também 28, o transmissor estará em *congestion avoidance*.

O pacote 57, entretanto, também foi perdido e assim em t=0,51 s são recebidos 25 ACKs duplicados para o 56, o que novamente dispara o *fast retransmit* e leva o transmissor a entrar em *fast recovery*. Portanto, o segmento 57 é retransmitido e novamente *ssthresh* e *cwnd* são reduzidos à metade (agora valem 14). A janela total vale 14+25=39 pacotes. No entanto, os segmentos de 57 a 135 transmitidos anteriormente continuam pendentes, totalizando 79. Neste momento não há dados armazenados no meio (exceto a retransmissão do 57), portanto os ACKs necessários para que o transmissor possa prosseguir simplesmente não serão recebidos.

De fato, em t=0,57 s chega o ACK do segmento 58, confirmando o sucesso da retransmissão do 57 e tirando o trasmissor do *fast recovery*. A janela de transmissão volta a receber o valor de *cwnd*, que é 14. Ainda temos 77 pacotes pendentes e nenhum em trânsito. A conseqüência disso é que o transmissor fica parado sem poder enviar nada até que ocorra um *time-out* e ele volte ao *slow start*.

Isto ocorre somente em t=0,88 s. Devido ao estouro do temporizador, o pacote 59 é retransmitido. A variável *cwnd* recebe o valor um porque estamos no início do *slow start*, e *ssthresh* volta a ser reduzida pela metade, passando a valer sete. Daí em diante, o transmissor segue no *slow start* até t=1,08 s, quando a *cwnd* alcança o valor de *ssth* entrando em regime de *congestion avoidance*.

O esgotamento do temporizador ocorre pelo fato de que o TCP Reno, para continuar "vivo" (enviando pacotes), necessita de 4 (quatro) reconhecimentos (3 duplicados) para cada pacote que foi descartado na janela, o que pode não ocorrer, como mostrado na simulação. Outra limitação notada é que o TCP Reno necessita de pelo menos um RTT para a retransmissão de cada pacote perdido.

IV.5.3 SACK

O gráfico da figura IV.11 mostra os resultados desta simulação para o TCP SACK.

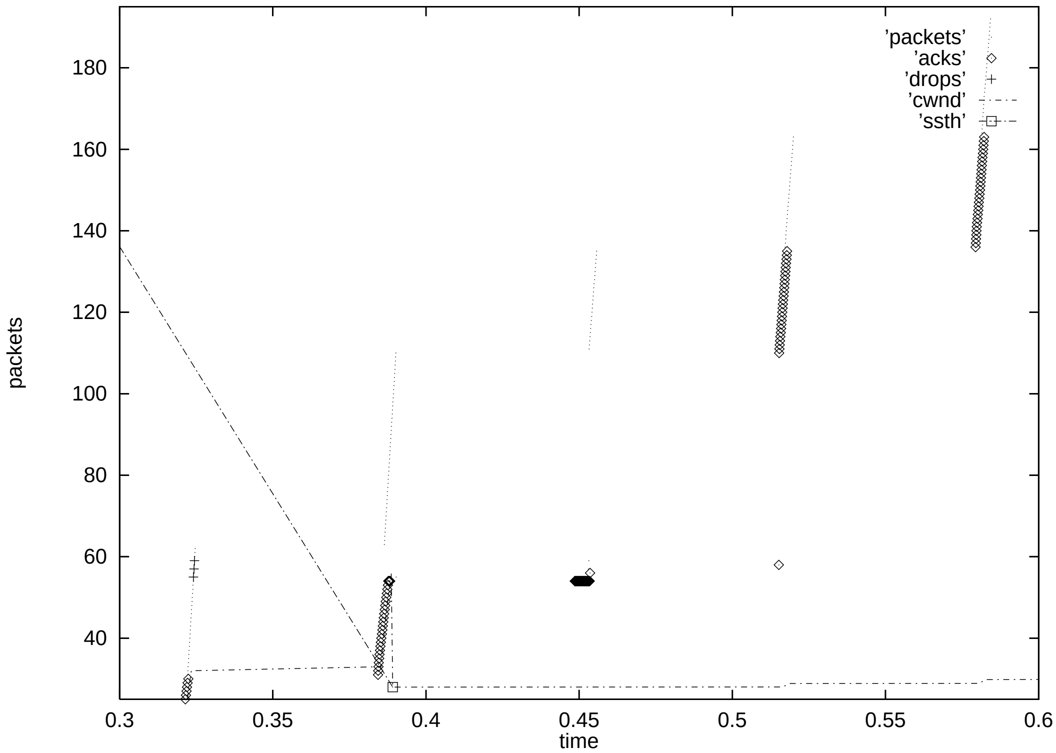


Figura IV.11: Três perdas com a utilização do TCP SACK

No caso seguinte temos três perdas em uma janela de transmissão, que foi enviada em t=0,32 s. Foram perdidos os segmentos de número 55, 57 e 59. É importante observar que o transmissor, ao enviar a janela até 62, ainda está em regime de *slow start*.

Em t=0,38 s, ou seja, passado 1 RTT após a transmissão dos segmentos da janela onde ocorreram as perdas, começam a chegar os ACKs relativos a eles. Como foi dito anteriormente, o transmissor está em *slow start*, logo para cada ACK recebido podem ser transmitidos dois segmentos. Conforme os ACKs são recebidos, começa a transmissão de uma nova rajada de dados. Como as perdas ocorreram no final da janela anterior, a transmissão de novos dados transcorre normalmente até o momento em que é recebido o terceiro ACK duplicado relativo ao segmento 55. No total, são recebidos 6 ACKs duplicados relativos à janela anterior. No momento em que isto ocorre, já foram enviados os pacotes de 63 a 110 (48 pacotes), e a variável *cwnd* tem valor 56. O recebimento do terceiro ACK duplicado força a retransmissão do 55 e faz com que o transmissor entre em *fast recovery*. A variável *cwnd* é reduzida à metade do seu valor anterior (passa a ser 28). A janela de transmissão passa então a ser min($awnd$, $cwnd$+$ndup$). Os valores destas variáveis são: $awnd$=500, $cwnd$=28 e $ndup$=6. A janela é portanto 34. Como há 48 segmentos enviados que ainda não foram reconhecidos, a janela precisa chegar a 49 para que se possa transmitir novos dados. Assim, após 15 novos ACKs duplicados recebidos, o transmissor poderá efetuar suas retransmissões.

Em t=0,45 s, são recebidos os reconhecimentos relativos à janela de 63 a 110. Como o pacote 55 ainda não foi recebido, estes ACKs são todos ACKs duplicados relativos àquele segmento. No total são recebidos 48 ACKs duplicados. Em seguida é recebido um ACK relativo ao segmento 56 (em resposta à retransmissão com sucesso do 55). Primeiramente vamos analisar a resposta aos ACKs duplicados. Como foi dito anteriormente, quando este número chega a 15, o transmissor já pode enviar mais dados. Como estamos usando a opção de reconhecimentos seletivos, o transmissor tem todas as informações dos segmentos que ainda não foram recebidos. Deste modo, quando for permitido que ele transmita, ou seja, assim que ele receber o número suficiente de ACKs para "inflar" a janela de transmissão, será efetuada a retransmissão do segmento 57. Quando o próximo ACK duplicado é recebido, a janela aumenta de mais um segmento e é retransmitido o 59. Os ACKs duplicados seguintes já provocam a transmissão de novos pacotes de dados (de 111 até 135). Depois dos ACKs duplicados, é recebido um reconhecimento que inclui o segmento 55, o que tira o transmissor do modo *fast recovery*, colocando-o em *congestion* *avoidance*. Neste ponto, a janela de transmissão volta a ser min(*awnd*, *cwnd*), sendo que *cwnd* recebe o valor de *ssthresh*. No caso, *cwnd* passa ser 28. Como o número de pacotes já transmitidos e não reconhecidos é maior que 28, o transmissor pára de enviar novos pacotes até que aqueles sejam reconhecidos.

Em t=0,52 s, chegam os reconhecimentos cumulativos dos segmentos até 110, e da nova janela de dados (111 em diante). A partir daí, o transmissor passa a operar em modo *congrestion avoidance*, estando inteiramente recuperado das perdas ocorridas.

A partir desta observação podemos concluir que o TCP SACK demonstra grande robustez em face de um número relativamente grande de perdas dentro de uma janela. O fator determinante nesta robustez é a eficiência nas retransmissões. Podemos ver que nenhum segmento foi retransmitido desnecessariamente (sem ter sido perdido). Sem o reconhecimento seletivo, o TCP teria duas opções: retransmitir indiscriminadamente, o que poderia piorar o congestionamento na rede; ou esperar um RTT para se informar sobre cada bloco perdido, o que aumentaria em muito o tempo de recuperação, além de ser ineficiente quanto ao uso do meio de transmissão (os efeitos são piores quanto maior a latência do caminho). Com o SACK, a recuperação foi muito rápida (aproximadamente dois RTTs) e foi evitado o time-out e, por conseguinte, a necessidade de fazer um novo *slow start*.

Cada ACK duplicado gerado carrega informações de pacotes recebidos fora de ordem no receptor, o que provê uma grande quantidade de informações ao transmissor. É importante lembrar que cada ACK duplicado deve necessariamente conter informações sobre o bloco fora de ordem que o deflagrou, ou seja, contém a informação mais recente sobre o que foi recebido. Portanto, apesar de o campo de SACK só ter espaço para especificar até três blocos de dados, conforme vão chegando mais segmentos fora de ordem o receptor vai enviando mais blocos, e essa informação fica guardada no transmissor para ser usada posteriormente, no momento da retransmissão.

IV.6 Avaliação de Eficiência Através da Transmissão de um Arquivo

Para avaliar a maior robustez do TCP SACK em relação ao TCP Reno, foi proposta a análise do tempo gasto com o envio de um arquivo, utilizando-se ambos os protocolos. Para tanto, empregou-se mais uma vez o simulador de redes *ns*. A topologia utilizada, mostrada na figura IV.12, apresenta uma fonte FTP que envia um arquivo de 666 pacotes de 1500 bytes (ou 999 Kbytes). A fonte está conectada a uma rede de 10 Mbps com latência de 1 ms, assim como o destino do FTP. Estas redes são interconectadas através de um enlace de 1 Mbps com latência variável. Os valores da latência do enlace de gargalo buscam retratar redes metropolitanas (30 ms), redes de longa distância (100ms) e enlaces de satélite (290 ms). A fila no roteador R0, na entrada do enlace mais lento, é de 13 pacotes de 1500 bytes e a *awnd* do receptor é de 200 pacotes, também com 1500 bytes cada.

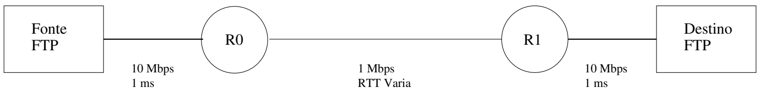


Figura IV.12: Topologia utilizada para verificação do desempenho dos TCPs Reno e SACK.

Esta topologia foi escolhida para que se pudesse observar claramente como o TCP busca a ocupação máxima do canal. Optou-se pela utilização de apenas uma fonte TCP, para que se soubesse exatamente a capacidade do canal em qualquer instante, sem a interferência de outros tráfegos. Assim é possível observar com qual eficiência os fluxos TCP, quando em equilíbrio, ocuparam o canal.

Em uma primeira análise devemos notar que quanto maior o RTT da rede, pior será o desempemho dos protocolos, no que diz respeito ao tempo total de transmissão. Este fato pode ser observado na figura IV.13. A vazão obtida na transmissão também é degradada da mesma forma, como se verifica através da figura IV.14. Nota-se, entretanto que a degradação no tempo necessário para a transmissão é maior no TCP Reno. Isto se dá por causa da necessidade de espera por *timeouts* para a recuperação do congestionamento, no caso da existência de muitas perdas. Lembramos aqui que o tempo de espera pelo *timeout* é proporcional ao RTT.

Como o TCP SACK consegue se recuperar das perdas, na maioria das vezes, sem a necessidade de esperar pelo *timeout*, podemos observar uma maior eficiência em relação ao Reno, principalmente para RTTs maiores.

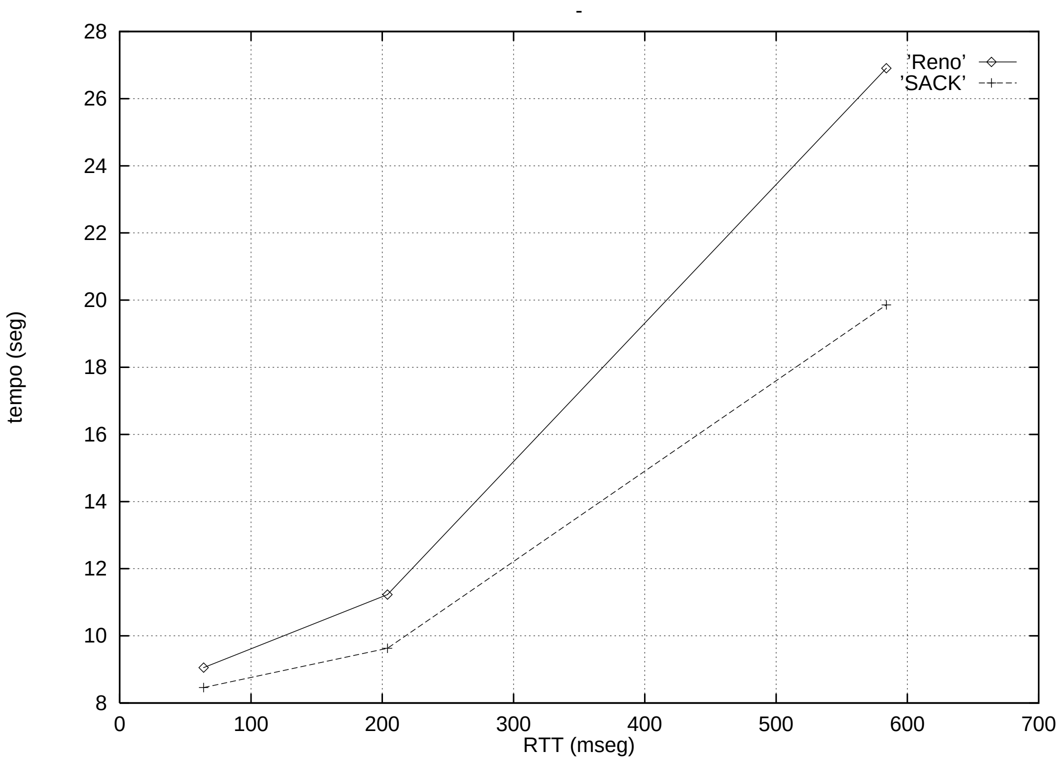


Figura IV.13: Tempo de transmissão do arquivo para diferentes latências.

É importante destacar que o melhor desempenho do TCP SACK ocorre graças ao mecanismo de reconhecimentos que permite o reenvio de pacotes de forma mais eficiente do que no TCP Reno, no caso de múltiplas perdas. Deve ser observado que os algoritmos de controle de congestionamento são os mesmos em ambas as implementações do protocolo. Desta forma, a ocupação do canal no estado estacionário da conexão será a mesma nos dois casos. De fato as medidas realizadas, apresentadas na tabela IV.1, demonstraram este fato.

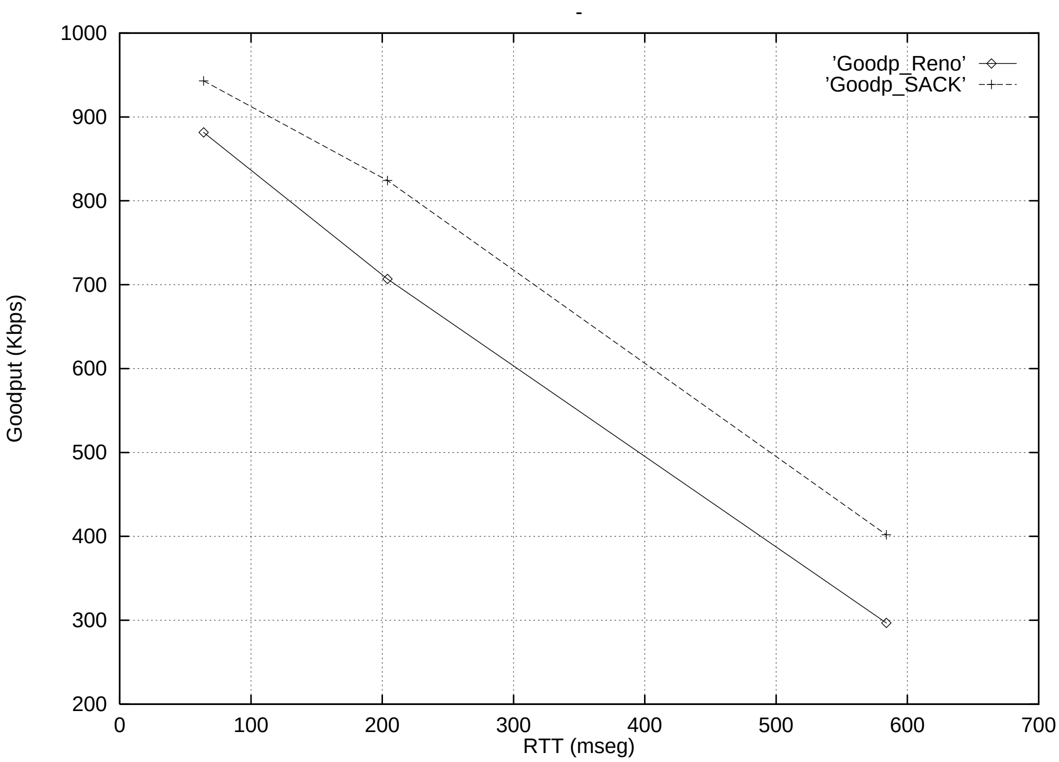


Figura IV.14: Vazão líquida obtida pelas diferentes implementações do TCP na transmissão do arquivo.

Tabela IV.1: Vazão obtida no estado estacionário para diferentes tempos de propagação.

TCP	30 ms	100 ms	290 ms
Reno	1,060 Mbps	1,042 Mbps	0,624 Mbps
SACK	1,066 Mbps	1,037 Mbps	0,636 Mbps

Para a obtenção da vazão no estado estacionário observou-se o gráfico de pacotes enviados no tempo, escolhendo-se uma região livre de perdas e suficientemente afastada do último *slow-start*, calculando-se o coeficiente angular da reta encontrada.

Como dito, a eficiência dos protocolos em termos de capacidade de ocupação do canal de comunicação é equivalente, uma vez que os algoritmos que atuam nesse sentido são os mesmos. Nota-se que o canal consegue ser plenamente ocupado nas latências menores, porém no caso onde o RTT fica na ordem de 580 ms os protocolos passam a encontrar dificuldades. Pela fórmula descrita na equação II.2 podemos observar os limites teóricos para cada caso. Assim quando o RTT é da ordem de 60 ms, o limite para a vazão fica em 5 Mbps. Já com RTT em 200 ms é possível que se alcance uma taxa de transmissão de até 1,5 Mbps. Já com um RTT maior, como 580 ms, a vazão fica limitada ao patamar de 0,517 Mbps.

Uma vez que pudemos observar que a ocupação do canal no estado estacionário da conexão não é dependente da versão de protocolo utilizada, fica claro que a maior eficiência do TCP SACK em relação ao TCP Reno ocorre de fato na fase transitória de recuperação de perdas, que é muito mais robusta e rápida na implementação com SACK.

IV.7 Análise das Simulações

A partir das simulações podemos verificar deficiências nas versões mais antigas do TCP, devido à utilização do esquema de reconhecimentos cumulativos. Devido a essas limitações, o TCP SACK buscou uma melhoria na eficiência do protocolo através da política de reconhecimentos seletivos, onde o receptor informa com precisão quais blocos de dados deixaram de ser recebidos. Assim, o emissor passa a dispor de uma maior quantidade de informações, de modo que possa otimizar as retransmissões.

Um ponto que se mantém crítico no TCP, em todas as versões simuladas, e que força a espera por esgotamento de temporização, é a relação entre o número de perdas e o tamanho da janela de transmissão. Como o TCP usa os ACKs como *clock* para a transmissão (e retransmissão) de segmentos, se o número de perdas for muito grande em relação à janela não haverá ACKs em número suficiente para que sejam executadas as retransmissões. Este problema se acentua muito em situações onde a janela é pequena (até 10 MSS) [22].

Mais especificamente, qualquer versão do TCP necessita de pelo menos três ACKs duplicados (ou seja, quatro ACKs no total) para executar o *fast retransmit*. Isto quer dizer que se houver por exemplo três perdas em uma janela de seis segmentos, independentemente da ordem dessas perdas, o *fast retransmit* jamais será disparado. Neste caso, não haverá alternativa senão esperar o temporizador.

No caso do Tahoe, o *fast retransmit* é seguido do *slow start*, portanto ele trata este instante como o início de uma conexão e não depende da chegada de mais reconhecimentos. Entretanto, o TCP Reno e suas modificações (New Reno e SACK) entram em modo *fast recovery*. Após a entrada nessa fase, a janela é inicialmente reduzida à metade e inflada de um segmento a cada ACK duplicado. Desta forma, o emissor precisa esperar pelo recebimento de meia janela de ACKs para conseguir efetuar uma nova transmissão. Isto porque ao notar uma perda, uma quantidade de pacotes equivalente a uma janela já foi enviada em avanço. A janela de transmissão nesse momento, entretanto, é reduzida à metade. Dessa forma, existem mais pacotes pendentes do que a janela permite ($cwnd/2$), e novos pacotes só poderão ser transmitidos quando a janela inflada for maior que a quantidade de pacotes pendentes (como já tinham sido enviados $1{*}cwnd$ pacotes e a janela estava em $1/2{*}cwnd$, é necessário que se infle $1/2{*}cwnd$ para novas transmissões).

O TCP Tahoe adota como estratégia realizar um *slow start* toda vez que ocorre uma perda. Isso implica no esvaziamento do canal mesmo em situações de congestionamento moderado, em que não há necessidade de uma reação tão radical. Assim, a eficiência do Tahoe fica muito comprometida. Quanto maior o RTT da conexão, maior será a degradação da vazão, pois o aumento da janela demora mais a ocorrer. Quando a janela chegar a *ssth*, o Tahoe entra em *congestion avoidance*. Entre o instante da perda e a volta ao estado estacionário, teremos $log_2(ssth)$-1 RTTs. Também pode ser observado que nas situações onde houve mais de uma perda na mesma janela, o TCP Tahoe retransmitiu pacotes desnecessariamente. Nas simulações para duas perdas por exemplo, houve uma retransmissão de pacote já recebido. Com três perdas, esse número aumentou para três.

O TCP Reno tem como opção a retransmissão de um pacote perdido em cada RTT, que é o tempo necessário para a chegada de reconhecimentos após cada retransmissão, trazendo novas informações ao emissor. O TCP Reno inclui o mecanismo de *fast recovery*, criado para evitar o esvaziamento do canal em situações do congestionamento moderado e assim melhorar esse comportamento em relação ao Tahoe. De fato, o Reno solucionou algumas limitações do Tahoe, mas por outro lado introduziu novos problemas. O TCP Reno é otimizado para o caso de uma perda por janela. Conforme o número de perdas aumenta, seu desempenho degrada rapidamente. Isso é causado pelo fato de o Reno executar o *fast retransmit* e entrar em *fast recovery* a cada perda verificada. Este fato acaba por reduzir a janela de transmissão pela metade repetitivamente, tantas vezes quanto forem os pacotes descartados em uma janela. Desta forma o TCP Reno acaba por reduzir a janela muito mais do que seria necessário para contornar uma situação de congestionamento (as perdas foram todas geradas no mesmo instante do congestionamento, onde a redução da *cwnd* pela metade já seria suficiente). Este fato atrapalha em muito o desempenho do Reno, uma vez que ao sair do *fast recovery* ele passa ao *congestion avoidance*, onde o crescimento da janela será linear, tornando muito lento o retorno da janela ao seu tamanho ideal.

No caso de uma perda, o Reno leva 1 RTT para se recuperar, e *cwnd* é reduzida à metade do valor inicial. Se ocorrem duas perdas, ele levará 2 RTTs e cwnd é reduzida a um quarto. Nesta situação, a recuperação do Reno é mais rápida que a do Tahoe (a menos que a janela inicial seja menor que 8, o que forçaria um estouro no temporizador). Contudo, ele entra em *congestion avoidance* com *cwnd* igual à metade do valor da *cwnd* do Tahoe. Esta situação já começa a ficar desfavorável para o Reno. Com três perdas o Reno levaria 3 RTTs para se recuperar, no entanto nestes casos, quase sempre ocorre um estouro no temporizador [3]

Já o TCP SACK, consegue alcançar o desempenho ótimo para um TCP com *fast recovery*. O SACK consegue se recuperar em 2 RTTs de qualquer situação em que o número de perdas seja menor que um quarto da janela. Para compreender isto nota-se que o SACK utiliza o *fast retransmit*, e assim como o Reno, necessita receber uma quantidade de ACKs equivalente à meia janela para inflar o *cwnd* e continuar a transmissão, como explicado acima. Dessa forma, para inflar a janela o suficiente para a retransmissão de todos os pacotes perdidos, será necessária a chegada de $(W/2 + D)$ ACKs (sendo W o tamanho da janela e D a quantidade de pacotes descartados) de modo que o SACK consiga se recuperar, retransmitindo todos os segmentos perdidos, em 2 RTTs. Essa

[3] No caso de 3 perdas numa mesma janela para o TCP Reno, haverá estouro do temporizador sempre que o número de pacotes entre a primeira e a segunda perda é menor que $2 + 3W/4$, sendo W o tamanho da *cwnd* logo antes do *fast retransmit* [20].

quantidade de pacotes, $(W/2 + D)$, é portanto o número de pacotes transmitidos que precisam chegar com sucesso ao receptor. Assim o número de pacotes que podem ser perdidos é $W - (W/2 + D) = W/2 - D$. Desta forma, fazendo $D = W/2 - D$, igualando o número de perdas ao máximo permitido, teremos D= W/4, que é o número máximo de perdas para um dado tamanho de janela. Assim, se o número de ACKs for metade da janela (e maior que 4), O SACK ainda consegue manter-se ativo, porque o segmento que foi transmitido no *fast retransmit* mantém os ACKs chegando (mesmo que seja um de cada vez).

Mesmo em casos onde o número de perdas chega a metade da janela ele ainda pode se recuperar, levando no máximo x RTTs, onde x é o número de perdas. Para as duas afirmativas acima, é necessário atender a condição de que haja ACKs duplicados suficientes para disparar o *fast retransmit*.

Tabela IV.2: Comparação da recuperação entre TCP Tahoe, Reno e SACK.

TCP	RTTs	Redução da *cwnd*
Tahoe	$log_2(cwnd)$-1	1/2
Reno	Perdas	$1/2^{Perdas}$
SACK	$\leqslant$2, Se possível	1/2

A tabela IV.2 traz uma comparação entre as três versões do TCP simuladas. Na segunda coluna está apresentado o tempo em RTTs, a partir do reenvio do pacote que foi descartado, que o protocolo leva para voltar à fase de *congestion avoidance*, desde que não haja estouro do temporizador. A terceira coluna mostra a razão de redução da janela ($cwnd_{novo}/cwnd_{antigo}$). Para o TCP Tahoe, o tempo de recuperação só é função do tamanho da janela no momento em que a perda foi detectada. Isso introduz robustez, uma vez que a quantidade de perdas não influencia na recuperação. Não há, entretanto, nenhuma tentativa de otimização para situações específicas. Outro fator de robustez é o fato da razão de redução da *cwnd* ser constante. Já para o TCP Reno ambos os valores são função do número de perdas. Isto demonstra que o Reno apresenta problemas quando as perdas aumentam, indicando sua pouca robustez. Observa-se que a recuperação para o TCP SACK ocorrerá em menos de 2 RTTs desde que seja respeitada a relação entre perdas e janela explicada anteriormente.

Capítulo V

Conclusões

O trabalho apresentado realizou uma análise do protocolo da camada de transporte da Internet, o TCP. Inicialmente esse protocolo foi abordado de uma maneira geral, onde se sumarizou o seu funcionamento, com o objetivo de criar o contexto para o foco principal a ser abordado, o controle de congestionamento. Em seguida, detalhou-se os principais mecanismos de controle de congestionamento, utilizados para o controle fim-a-fim do TCP.

Para aprofundar a compreensão dos mecanismos explicados, realizaram-se simulações de um cenário típico de rede, com um enlace de gargalo. Variando-se as características da rede obtivemos diversos padrões de descarte de pacotes. As simulações permitem uma análise qualitativa do desempenho dos protocolos. A partir das mesmas podemos concluir em linhas gerais:

- O TCP Tahoe é a implementação mais conservadora dentre as simuladas. Ele não busca otimizar situações específicas de perdas, apresentando assim grande robustez e estabilidade. Em contrapartida, nem sempre seu comportamento propicia uma boa vazão. Além disso, verifica-se que o Tahoe efetua transmissões desnecessárias com freqüência.
- O TCP Reno apresenta melhoria de desempenho em relação ao Tahoe nos casos

de congestionamento leves, com no máximo um descarte por janela de transmissão. Por ser um pouco mais agressivo que o Tahoe, ele é mais instável, o que leva a situações onde seu desempenho fica bastante prejudicado.

- O TCP SACK, devido a sua política de reconhecimentos incrementada, apresenta maior eficiência, que se explica pela maior quantidade de informações disponíveis para a realização das retransmissões.

Pode se observar que uma das principais preocupações demonstradas no projeto das versões do TCP é evitar que ocorram estouros dos temporizadores. Este é um ponto crítico por dois motivos. Em primeiro lugar, o estouro de temporizador significa o esvaziamento do canal, o que não é desejável pois reduz a vazão e a eficiência de utilização do canal. Soma-se a isso o problema da resolução do relógio empregado pelo TCP. Apesar da estimativa do RTT ser feita utilizando algoritmos bastante refinados, como mostrado na seção III.2.1, a baixa resolução do relógio implica em pouca exatidão dos dados que alimentam os algoritmos. Portanto, o RTT utilizado no cálculo do temporizador será quase sempre superestimado, forçando uma longa espera pelo estouro do temporizador, o que faz com que o canal fique desocupado por mais tempo, implicando em maiores prejuízos para a vazão da conexão. Essa situação pode ser verificada nas simulações do TCP Reno no cenário com três perdas.

As simulações realizadas ilustram com clareza as limitações no desempenho das versões mais antigas do TCP (Tahoe e Reno) e como o SACK consegue superar esses problemas. Na realidade, observa-se que uma conseqüência da não utilização de reconhecimentos seletivos é que o transmissor deve optar por uma das seguintes estratégias para a recuperação dos pacotes descartados:

1. Retransmitir pacotes que já foram possivelmente entregues, realizando retransmissões desnecessárias, como no TCP Tahoe.

2. Retransmitir no máximo um pacote por RTT, como no caso do TCP Reno e New Reno.

A fragilidade do Reno para o caso de ocorrerem múltiplas perdas em uma janela de transmissão pode ser corrigida pelo New Reno, que mais uma vez vem acrescentar novas informações ao emissor. Entretanto, a ausência de reconhecimentos seletivos não permite uma recuperação ótima, forçando a retransmissão de no máximo um segmento por RTT.

A introdução da política de reconhecimentos seletivos no TCP SACK solucionou muitos dos problemas observados nas demais implementações. Ao receber reconhecimentos duplicados ele infere que houve perdas, assim como nos outros TCPs. Devido às informações mais completas sobre os descartes ocorridos, entretanto, ele consegue aproveitar melhor as retransmissões, enviando apenas os pacotes necessários e não necessitando da espera pela chegada de novos ACKs para a obtenção do estado completo do *buffer* do receptor.

A grande desvantagem para a introdução do esquema de reconhecimentos seletivos no TCP é a necessidade de modificações tanto nos emissores como nos receptores, o que não ocorre nas propostas do Reno, New Reno e Vegas. Assim, estas últimas propostas, mesmo que não sejam ótimas são bem mais viáveis no escopo atual da Internet.

Referências Bibliográficas

[1] G. Miller e K. Thompson, "The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone", tech. rep., Cooperative Association for Internet Data Analysis, 1998.

[2] "Internet Domain Survey", tech. rep., Internet Software Consortium, janeiro 2000. URL: `http://www.isc.org/ds/`.

[3] V. Jacobson e M. J. Karels, "Congestion Avoidance and Control", em *Proceedings of ACM SIGCOMM, Symposium on Communication Architectures and Protocols*, pp. 314–329, 1988.

[4] W. Stevens, "TCP Slow start, Congestion Avoidance, Fast Retransmit and Fast Recovery Algorithms", *Internet RFC 2001*, janeiro 1997.

[5] V. Jacobson, "Modified TCP Congestion Avoidance Algorithm", tech. rep., E-mail to the end2end-interest mailing list, 30th. Apr. 1990. URL: `ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt`.

[6] S. Floyd e T. Henderson, "The New Reno Modification to TCP's Fast Recovery Algorithm", *Internet RFC 2582*, abril 1999.

[7] J. S. Ahn, P. Danzig, Z. Liu e L. Yan, "Evaluation of TCP Vegas: Emulation and Experiment", em *Proceedings of ACM SIGCOMM, Symposium on Communication Architectures and Protocols*, 1995.

[8] M. Mathis, J. Mahdavi, S. Floyd e A. Romanow, "TCP Selective Acknowledgment Options", *Internet RFC 2018*, outubro 1996.

[9] K. Fall e K. Varadhan, "Network simulator - NS (version 2) notes and documentation", tech. rep., The VINT Project, julho 1999. URL: http://www-mash.cs.berkeley.edu/ns/ .

[10] L. L. Peterson e B. S. Davie, *Computer Networks: A Systems Approach.* San Francisco, California: Morgan Kaufmann Publishers, 1996.

[11] I. B. H. A. Alves e R. R. Brasil, "Análise de Desempenho de Protocolos TCP", relatório técnico, GTA/COPPE/PEE - UFRJ, 1999.

[12] R. Jain, "Congestion Control in Computer Networks: Issues and Trends", *IEEE Network Magazine*, vol. 4, no. 3, pp. 24–30, maio 1990.

[13] S. Floyd e V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, agosto 1993.

[14] K. Nichols, S. Blake, F. Baker e D. L. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", *Internet RFC 2474*, dezembro 1998.

[15] S. Blake, D. L. Black, M. Carlson, E. Davies, Z. Wang e W. Weiss, "An Architecture for Differentiated Services", *Internet RFC 2475*, dezembro 1998.

[16] J. Postel, "Transmission Control Protocol", *Internet RFC 793*, setembro 1981.

[17] D. E. Comer, *Internetworking with TCP/IP*, vol. 1. New Jersey: Prentice Hall, 1995.

[18] A. S. Tanenbaum, *Computer Networks.* New Jersey: Prentice Hall, 3rd. ed., 1996.

[19] W. R. Stevens, *TCP/IP Ilustrated, Volume 1: The Protocols.* Addison-Wesley Publishing Company, 1994.

[20] S. Floyd e K. Fall, "Simulation-based Comparisons of Tahoe, Reno, and Sack TCP", *ACM Computer Communications Review*, vol. 26, no. 3, pp. 42–48, julho 1996.

[21] L. S. Brakmo e L. L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a global internet", *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, pp. 1465–1480, outubro 1995.

[22] D. Lin e H. T. Kung, "TCP Fast Recovery Strategies: Analysis and Improvements", em *Proceedings of INFOCOM*, San Francisco, CA, pp. 263–271, março 1998.

[23] M. Aron e P. Druschel, "TCP: Improving Startup Dynamics by Adaptative Timers and Congestion Control", tech. rep., Rice University, 1998.

[24] M. Allman, V. Paxon e W. Stevens, "TCP Congestion Control", *Internet RFC 2581*, Abril 1999.

[25] H. M. Chaskar, T. V. Lakshman, U. Madhow, "TCP Over Wireless with Link Level Error Control: Analysis and Design Methodology", *IEEE/ACM Transactions on Networking*, pp. 605–615, outubro 1999.

Apêndice A

Simulador de Redes

Neste apêndice o ambiente do simulador de redes *ns* é brevemente apresentado. Uma listagem típica de um *script* OTcl é mostrada e comentada. Alguns detalhes importantes das implementções do TCP no *NS* que afetam diretamente as simulações são também aqui colocados.

A.1 O Ambiente do Simulador *NS*

O simulador é um software orientado à eventos, escrito em C++, utilizando a linguagem de script orientada a objeto OTcl para a interface com o usuário. A utilização de linguagens diferentes permite mais flexibilidade ao simulador. Se o usuário estiver simulando novos protocolos, cujas rotinas complexas possam gastar muito tempo de processamento, ele pode utilizar-se da eficiência e da robustez do C++, na manipulação de bytes e de cabeçalhos de pacotes por exemplo. O C++ oferece tempos de execução baixos, porém a depuração pode ser mais lenta.

O OTcl, que é utilizado para a configuração dos cenários de simulação apresenta características boas para esta função. Diferentemente do usuário que busca a implementação de um protocolo complexo, aquele que busca analisar os efeitos de modificações de parâmetros nas redes necessita de facilidade e rapidez na configuração de cenários. Como a configuração de um cenário é realizada apenas no início da simulação, o tempo de execução não passa a ser um fator crítico, sendo a simplicidade da interação com o usuário o aspecto mais importante.

Desta forma os mecanismos simulados são escritos em C++, enquanto que os cenários de simulação são configurados através do OTcl. As simulações realizadas neste trabalho utilizaram as implementações de TCP distribuídas em conjunto com a versão 2 do *NS*. Os cenários foram criados através de scripts OTcl.

O script utilizado neste trabalho é um bom exemplo de configuração de cenário para o simulador. Inicialmente devemos instanciar um novo simulador:

```
set ns [new Simulator]
```

O comando mostrado acima cria uma variável `ns` que é associada ao novo simulador. Em seguida, devemos definir os arquivos utilizados para o armazenamento dos dados:

```
set f [open out.tr w]
set nf [open out.nam w]
set tcpfile [open tcp.tr w]
```

Após este passo, a topologia a ser simulada deve ser criada. Um procedimento pode ser definido com esta finalidade. O primeiro passo inclui a criação dos nós da rede:

```
#Procedure para criar a topologia
proc createtopology {} {
    global ns n s f teste
    set qlim 13

    for {set i 0} {$i<=1} {incr i} {
```

```
        set n($i) [$ns node]
    }
    for {set i 0} {$i<=15} {incr i} {
        set s($i) [$ns node]
    }
```

Em seguida, os enlaces entre os nós são definidos, com as características de retardo, vazão, tipo e tamanho das filas nos nós de entrada, entre outros parâmetros.

```
    for {set i 0} {$i<=7} {incr i} {
        $ns duplex-link $s($i) $n(0) 100Mb 1ms DropTail
        $ns duplex-link $s([expr $i + 8]) $n(1) 100Mb 1ms DropTail
    }
    $ns duplex-link $n(0) $n(1) 155Mb 30ms DropTail

    set fifoq [[$ns link $n(0) $n(1)] queue]
    $fifoq set limit_ $qlim
```

Por fim, os parâmetros a serem monitorados devem ser definidos e associados aos respectivos arquivos, que foram abertos anteriormente:

```
    $ns trace-queue $n(0) $n(1) $f
    $ns trace-queue $n(0) $s(0) $f
```

A próxima etapa constitui-se da definição do procedimento para a criação dos geradores de tráfego, através da escolha das fontes e destinos, e de suas associações aos respectivos nós.

```
#Procedure para criar e associar as fontes TCP
proc createfontes {} {
    global ns n s tcp endsim tcpfile psize init
```

```
set psize 500
set adv_wnd 500

for {set i 0} {$i<=7} {incr i} {
    set tcp($i) [$ns create-connection TCP/Sack1 $s($i)
                            TCPSink/Sack1 $s([expr $i + 8]) $i]
    $tcp($i) set packetSize_ $psize
    $tcp($i) set window_ $adv_wnd
    set ftp($i) [$tcp($i) attach-source FTP]
}
```

Ao se escolher a fonte, indica-se o protocolo utilizado, assim como parâmetros de sua configuração. O tipo de tráfego gerado também é definido. Neste exemplo, foi utilizado o protocolo TCP com a opção por reconhecimentos seletivos, o SACK. Foi definido o tamanho dos pacotes em bytes, o tamanho da *awnd* em pacotes e o tipo de tráfego, que foi FTP.

Em seguida, foram definidos os instantes de início e de fim da atividade das fontes FTP:

```
for {set i 0} {$i<=7} {incr i} {
    $ns at [expr 0.0+ $i/10000.0] "$ftp($i) start"
    $ns at [expr $endsim - 1.0] "$ftp($i) stop"
}
```

Por fim, um procedimento de finalização de simulação deve ser criado.

```
#Define a 'finish' procedure
proc finish {} {
    global ns f nf tcpfile prate riotrace teste tcp endsim psize tempo
    $ns flush-trace
```

```
close $f
close $nf
close $tcpfile

for {set i 0} {$i<=7} {incr i} {
    set ack [$tcp($i) set ack_]
    set goodp [expr $ack*8*$psize/($endsim - 1.0)/1000.0]
    puts "goodp TCP $i => $goodp Kbps"
}
```

Inicialmente, o procedimento fecha os arquivos com os dados obtidos. Em seguida a vazão líquida em cada conexão é calculada.

O próximo passo efetua uma filtragem nos dados obtidos, formatando os mesmos para que possam ser plotados diretamente por algum programa. No caso realiza-se uma formatação para a visualização com o software Gnuplot. Um script para o Gnuplot, nesse caso o `cmd.gnu`, também é criado e executado automaticamente.

```
set gpfile [open cmd.gnu w]
set awkCode {
    $1 != "-" { print; }
}
exec getrc -b -s 0 -f 0 out.tr | \
awk $awkCode | \
raw2gp -a -s 1 -m 500 -g > plot.gnu
exec grep cwnd tcp.tr > cwnd
exec grep ssthre tcp.tr > ssth
exec cat plot.gnu >@ $gpfile
puts $gpfile "set grid"
puts $gpfile "replot 'cwnd' u 1:7 w linesp 6"
puts $gpfile "replot 'ssth' u 1:7 w linesp 5"
```

```
    puts $gpfile "pause -1"
    close $gpfile
    exec rm plot.gnu
    exec gnuplot cmd.gnu
    exit 0
}
```

Por fim, o programa principal, que chama os procedimentos definidos, deve ser estruturado:

```
proc main {} {
    global ns endsim

    createtopology
    createfontes
    #Call the finish procedure after $endsim of simulation time
    $ns at $endsim "finish"
```

A última providência é a execução do comando `run`, que realiza a simulação de fato.

```
    #Run the simulation
    $ns run
}

main
```

Devemos ressaltar que o simulador trata os dados transmitidos em unidades de pacotes, e não de bits. Assim, o tamanho dos pacotes enviados é fixo, definido no início da simulação. Outro detalhe que não pode ser esquecido é que os procedimentos de conexão/desconexão das sessões TCP não são consideradas. Desta forma apenas os pacotes de dados e os respectivos reconhecimentos são enviados pela rede simulada.

Um outro ponto importante de se notar é o fato que a implementação do TCP SACK usada no *NS* mantém uma variável chamada *pipe* durante a etapa de *Fast Recovery*, que a diferencia da implementação do TCP Reno. Esta variável estima o número de pacotes presentes no meio de transmissão, e controla as retransmissões da seguinte forma: somente se pode transmitir um pacote se *pipe* for menor que *cwnd*. *Pipe* é inicializada com o valor de *cwnd-ndup*. É incrementada de um quando um segmento é transmitido (ou retransmitido) e decrementada de um quando um ACK reporta (através de SACK) que novos segmentos foram recebidos. Deve-se frisar que o decremento é feito para cada ACK com novos dados, não para cada segmento reportado, que seria uma política mais agressiva. Para ACKs parciais, *pipe* é decrementado de dois. A razão para isso é que um ACK parcial reconhece um segmento que foi retransmitido, o que implica que saíram do meio tanto o pacote transmitido originalmente (supostamente perdido) como o retransmitido. Esta algoritmo trata o problema de *quando* retransmitir, enquanto que o emissor constrói um quadro com os blocos SACKed, que trata de *quais* segmentos serão retransmitidos.