

Parte II

Transferência de dados

As aplicações distribuídas envolvem geralmente a transmissão de dados entre emissores e receptores, e por isso esta funcionalidade é fundamental nas redes de computadores. Na verdade, no modelo de camadas existe um nível inteiro dedicado a este objectivo, o nível de transporte. A importância de se discutirem diversas técnicas de transferência de dados tem por origem o facto de que quer os canais, quer a rede em geral, funcionarem com base na transferência de pacotes e, pelas mais diversas razões, estes podem não chegar ao seu destino, ou chegarem com erros, ou fora de ordem.

Apesar de ser bastante mais raro, os dados de uma mensagem podem ser corrompidos para além do seu transporte pela rede, como por exemplo na memória de um computador. As aplicações que não suportam este tipo de erros utilizam técnicas especiais para os detectar e corrigir mas a sua discussão ultrapassaria o âmbito deste capítulo. Neste capítulo vamos ver como é possível lidar com os erros com origem na rede.

Através das técnicas de detecção de erros discutidas na secção 2.4 é possível detectar se existem ou não diferenças de conteúdo entre a emissão e a recepção de um pacote. Apesar de existirem aplicações que podem conviver com pacotes com alguns erros, a prática mais frequente é suprimi-los antes de estes serem entregues ao destino. Por esta razão, raramente as aplicações recebem dados com erros ou, para efeitos práticos, a grande maioria das aplicações considera que a probabilidade de isso acontecer é, na prática, desprezável.

Assim, os erros considerados no contexto do transporte de pacotes de dados são as falhas de omissão, caracterizadas por os pacotes não chegarem ao destino, e ainda os pacotes entregues fora de ordem. Se estamos perante aplicações que não suportam essas falhas, é necessário compensá-las obtendo os pacotes que faltam e colocando-os na ordem adequada.

A necessidade de mascarar estas falhas pode aparecer a quase todos os níveis. Se um dado canal tem uma taxa de erros muito elevada, pode ser importante tentar corrigir logo a esse nível esses erros. Como ao nível de transporte existem protocolos de transferência fiável de dados, é obrigatória a correcção destas falhas por parte desses protocolos. Finalmente, quando uma aplicação se baseia em protocolos de transporte não fiável, e necessita de compensar estas falhas, o problema também se coloca. Teoricamente, o nível rede poderia também tentar corrigir este tipo de erros mas, devido ao princípio de privilegiar os extremos, essa opção não é geralmente utilizada.

Assim, nesta parte, logo no Capítulo 6, começaremos por discutir técnicas de compensação da ausência de pacotes baseadas na retransmissão dos pacotes em falta e técnicas de reordenação dos pacotes recebidos fora de ordem. No capítulo a seguir, o Capítulo 7, é discutida a utilização dessas técnicas no quadro do protocolo TCP.

As omissões de pacotes têm duas origens distintas: erros de transmissão nos canais e ritmos de chegada de novos pacotes que os comutadores não comportam. Sempre que pacotes são desprezados pelos comutadores por falta de espaço nas filas de espera, a rede está a desperdiçar recursos, *i.e.*, a capacidade de transmissão que esses pacotes já consumiram. Por outro lado, filas de espera muito grandes aumentam o tempo de trânsito, podendo levar a retransmissões inúteis, outro desperdício. O Capítulo 8 analisa formas de lidar com estes problemas e diminuir a probabilidade de se perderem demasiados pacotes por saturação da rede, *i.e.*, por se estar a utilizá-la acima das sua capacidade. Boa parte deste capítulo é dedicada ao estudo da forma como o protocolo TCP tenta encontrar um ritmo de emissão adequado, *i.e.*, um ritmo que minimize a perda de pacotes por saturação dos comutadores.

Em certos contextos é possível lidar com informação incompleta ou aproximada, tentando inferir a parte que falta. Os seres humanos são particularmente hábeis a recompor o significado de textos, sons ou imagens com pequenos erros por exemplo. Este facto pode ser aproveitado em certas aplicações que usam transferência de informação parcialmente errada, mas que recompõem, de forma aproximada, a parte

que falta. O Capítulo 9 apresenta algumas das técnicas usadas para compensação parcial de erros de omissão de pacotes. Estas técnicas são frequentemente usadas por algumas aplicações multimédia que conseguem lidar com informação incompleta ou de resolução inferior.

Finalmente, o Capítulo 10 apresenta uma breve referência a um conjunto de protocolos de transporte menos populares, ou emergentes, que apresentam formas alternativas de transferência de dados de forma fiável.

Capítulo 6

Fiabilidade com base em retransmissão

All things are difficult before they are easy.

– Autor: Thomas Fuller

Um fluxo de pacotes entre dois nós da rede é uma sequência de pacotes que está a ser transmitida do nó emissor para o nó receptor. O fluxo é fiável se todos os pacotes emitidos chegarem ao receptor, sem alterações e na ordem adequada. As técnicas de detecção de erros permitem rejeitar os pacotes que, para efeitos práticos, foram alterados entre a emissão e a recepção. Por outro lado, admite-se que a rede possa, por várias razões, perder pacotes, ou trocar a ordem da sua entrega. Assim, para garantir a fiabilidade do fluxo, é necessário que o receptor receba uma cópia sem erros de todos os pacotes emitidos e que consiga colocá-los na ordem adequada.

Neste capítulo vamos estudar formas de implementar a fiabilidade de um fluxo com base na detecção de quais os pacotes em falta, e de formas de levar o emissor a retransmitir uma nova cópia dos mesmos.

As hipóteses de trabalho são as seguintes. A rede é uma rede de comutação de pacotes sem estado, ver a Secção 4.1, que pode perder pacotes, e entregar pacotes ao destinatário por ordem diferente da sua emissão. O receptor recebe pacotes sem erros porque a interface dos canais rejeita os pacotes alterados no seu trajecto pelos mesmos. Os nós da rede não corrompem os pacotes entre a recepção e a emissão ou, se o fizerem, existem mecanismos de detecção, implementados entre o emissor e o receptor, que permitem identificar e rejeitar os pacotes alterados. Os canais e os nós da rede podem perder pacotes, mas não os perdem a todos, *i.e.*, mais tarde ou mais cedo alguns pacotes chegarão ao receptor. Sem esta última hipótese o problema não tem solução. Os pacotes com informação fluem do emissor para o receptor, mas pacotes com informação de controlo podem fluir do receptor para o emissor. Finalmente, os métodos usados devem tentar maximizar o débito extremo a extremo, ver 3.6.

Os protocolos a desenvolver podem ser usados no nível canal, para compensar os erros num canal com taxa de erros muito elevada, ou nos níveis transporte ou aplicação, para implementar fiabilidade na transmissão de dados. No primeiro caso não é necessário considerar a hipótese de que os pacotes podem chegar fora de ordem, pois, por hipótese, um canal não troca a ordem dos pacotes pois não contém *buffers* nem proporciona caminhos alternativos. Em ambos os casos é necessário considerar que é possível a comunicação nos dois sentidos, mesmo que sujeita a perda de pacotes.

A terminologia a usar para designar as unidades de informação transmitidas pelos protocolos, ver 4.4, poderia ser *frames*, pacotes, segmentos, blocos de dados ou

mensagens, dependendo do nível a que o protocolo é usado. Por comodidade, a seguir usaremos o termo pacotes, excepto quando o mesmo pode provocar confusão.

Para evitar introduzir complexidade inútil, vamos também começar por tentar encontrar soluções simples para o problema, e só adoptar soluções mais complexas se tal se revelar estritamente necessário.

6.1 Mecanismos de base

A solução mais simples consiste em o emissor emitir os pacotes em sequência à velocidade máxima que o canal o permitir. Essa solução tem dois problemas, ambos ilustrados na Figura 6.1. Por um lado não garante que todos os pacotes cheguem ao destino, e por outro, os pacotes podem chegar com um tal ritmo que o receptor não tenha tempo de os tratar em tempo útil. Nesta última situação, o receptor ficará numa situação equivalente à da rejeição de pacotes com erros pois não poderá processar todos os pacotes recebidos.

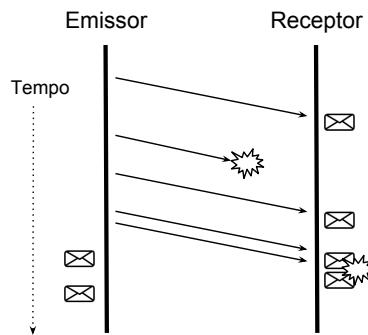


Figura 6.1: Diagrama temporal da transmissão de pacotes com informação do emissor para o receptor, com ilustração da perda de um pacote e da emissão de outro demasiado cedo

A segunda situação requer um mecanismo para adaptar a velocidade de emissão ao ritmo de processamento dos pacotes pelo receptor. Estes mecanismos designam-se mecanismos de controlo do fluxo.

Um **mecanismo de controlo de fluxo** (*flow control*) é um mecanismo de extremo a extremo que adapta o ritmo de emissão de pacotes pelo emissor à capacidade de o receptor os tratar em tempo útil.

A solução que podemos tentar a seguir consiste em obrigar o receptor a enviar ao emissor, após ter tratado cada pacote, um pequeno pacote de controlo indicando-lhe que pode emitir o pacote seguinte. A Figura 6.2 ilustra o funcionamento do mecanismo através de um diagrama temporal.

A Figura 6.3 mostra que esta solução tem pelo menos dois problemas, ambos relacionados com a perda de pacotes. Se se perderem pacotes do emissor para o receptor, ou do receptor para o emissor, o fluxo bloqueia.

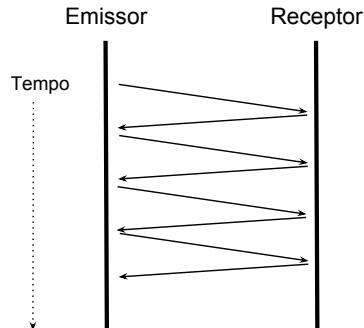


Figura 6.2: Diagrama temporal da transmissão de pacotes com informação do emissor para o receptor e de pacotes de controlo no sentido contrário

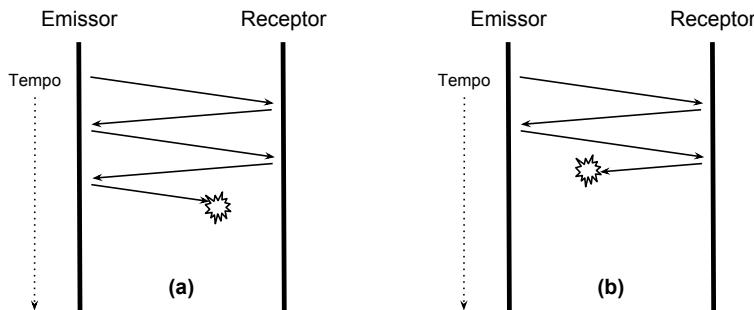


Figura 6.3: Bloqueio da transferência por (a) perda de um pacote com informação ou por (b) perda de um pacote de controlo

A solução para um problema deste tipo pode ser conseguida introduzindo um mecanismo que consiste na utilização de um **alarme temporizado (timer)**. O temporizador é inicializado com uma duração (d em segundos) e, caso não seja desarmado antes, depois de passarem d segundos sobre a sua inicialização, dispara um alarme. O momento do disparo do alarme é designado em inglês por *timeout time*, ou simplesmente *timeout*. Como o termo *timeout* é muito popular, vamos utilizá-lo frequentemente.

Assim, sempre que um pacote de dados é emitido, o emissor inicializa o temporizador com um valor adequado. Se um pacote de controlo chegar antes de o temporizador disparar um alarme, o temporizador é anulado, senão o alarme dispara e o último pacote emitido é retransmitido pelo emissor. O temporizador pode ser visto como uma espécie de relógio despertador, que é desligado pela chegada de um pacote e que toca caso nenhum chegue. A Figura 6.4 mostra como a utilização do alarme temporizado permite desbloquear o processo e fazê-lo recomeçar.

Infelizmente, ainda não temos uma solução correcta para o problema pois, como

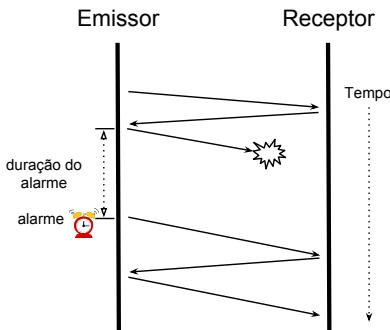


Figura 6.4: Utilização de um temporizador para desbloquear uma situação de bloqueamento por perda de pacotes

mostra a Figura 6.5 (a), a solução apresentada não trata correctamente a situação em que, ao invés de se perder o pacote com informação, o que se perde é o pacote de controlo. Nesta situação, a reemissão de pacotes de informação introduz duplicados no receptor, o que viola a correcção da solução.

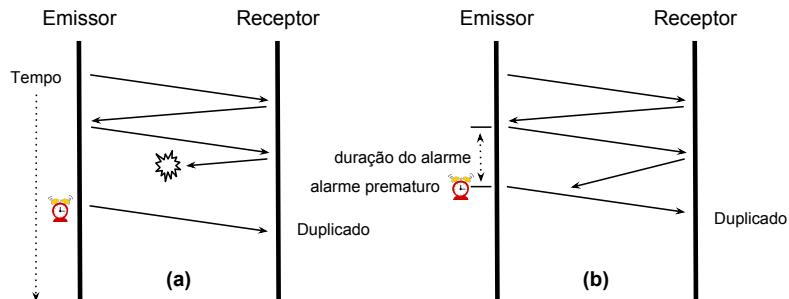


Figura 6.5: A perda de um pacote de controlo (a) ou um alarme prematuro (b) conduzem à recepção de um pacote em duplicado

Por outro lado, a introdução de um temporizador com alarme introduz um problema novo que consiste na regulação da sua duração. Com efeito, se o alarme ocorrer demasiado cedo, um pacote também pode ser emitido em duplicado, e nesse caso, o objectivo final também foi violado, ver a Figura 6.5 (b). Estas anomalias podem conduzir a situações como a ilustrada na Figura 6.6, em que o número de pacotes recebidos pelo receptor é igual ao número de pacotes emitidos, mas um foi duplicado e o outro está em falta.

A solução passa pela introdução de mais um mecanismo, que consiste na utilização de um número de sequência dos pacotes. A ideia consiste em numerar em sequência todos os pacotes, pois desta forma é possível detectar duplicados no receptor, e indicar exactamente ao emissor qual o pacote que foi de facto recebido pelo receptor. Assim,

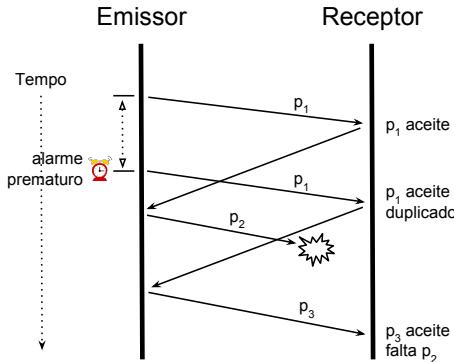


Figura 6.6: Duplicado de um pacote por o alarme ocorrer demasiado cedo, seguido da perda não detectada de um pacote

todos os pacotes emitidos contêm um cabeçalho com o número de sequência do pacote (por exemplo 1, 2, 3, ...), e os pacotes de controlo contêm o número de sequência do pacote cuja correcta recepção anunciam. Acresce, como se verá a seguir, que os números de sequência vão permitir resolver o problema de a rede poder entregar os pacotes por uma ordem diferente da com que foram emitidos.

O conjunto dos mecanismos até agora introduzidos, a saber, pacotes de dados, pacotes de controlo, temporizadores e números de sequência, são a base de um conjunto de protocolos, dos mais simples, aos mais sofisticados, que serão introduzidos a seguir. Esses protocolos baseiam-se na ideia da reemissão dos pacotes em falta e são por vezes designados por protocolos ARQ (*ARQ – Automatic Repeat ReQuest*).

Os protocolos de transferência fiável de dados em redes de pacotes, ou em canais, que se baseiam no método de retransmissão dos pacotes em falta, usam os seguintes mecanismos de base: pacotes de dados, pacotes de controlo, alarmes temporizados (*timeouts*) e números de sequência.

Para a realização destes protocolos são necessárias mensagens em que na parte de dados viajam os dados a transmitir e o cabeçalho contém a informação de controlo. A informação mínima necessária é a que consta da Figura 6.6, *i.e.*, um código de operação e um número de sequência.

O código de operação toma um dos seguintes valores: mensagem com dados, mensagem de confirmação, e também podem ser usadas mensagens de notificação de erro. A seguir usaremos as abreviaturas usadas frequentemente na literatura em língua inglesa para código de operação e que são muito populares nos protocolos normalizados.

DATA nas mensagens de dados

ACK nas mensagens de confirmação de recepção (**ACKnowledgement**)

NACK nas mensagens notificação de erro (**Negative ACKnowledgement**)

A Figura 6.8 mostra como a utilização de números de sequência nas mensagens de dados e de ACK resolve os problemas ilustrados na Figura 6.6, garantindo que o receptor recebe todos os pacotes sem faltas, e pela ordem correcta.

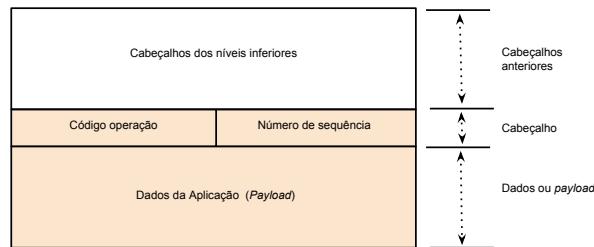


Figura 6.7: Formato das mensagens usadas pelos protocolos de transferência fiável de dados com base em retransmissão

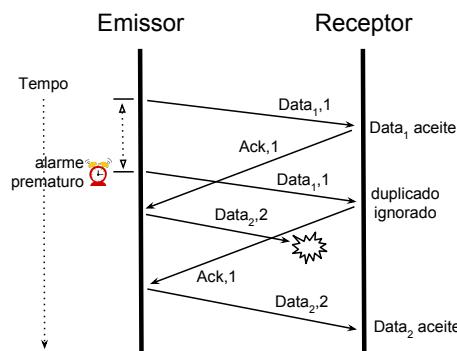


Figura 6.8: Utilização de números de sequência para garantir a correcta recepção dos pacotes

O campo com o código de operação pode ter um pequeno número de bits, um byte por exemplo parece ser suficiente. Mas quantos bits necessitamos para o campo com o número de sequência? À primeira vista poderia parecer que a dimensão em bits limitaria o número máximo de pacotes do fluxo. Por exemplo, se for usado um byte, então não podemos ter mais do 256 mensagens no fluxo. Na verdade, isso é falso, pois nada impede que se dê a volta, e se recomece a numerar as mensagens de novo com os números iniciais. Quanto maior for esse campo, maior será o desperdício, e por isso há todo o interesse em ter um campo relativamente pequeno.

No entanto, é preciso garantir que uma mensagem antiga não seja confundida com uma nova. Numa rede com comutadores que possam trocar a ordem dos pacotes, é necessário garantir que os números de sequência levam mais tempo a ser reutilizados que o tempo máximo de vida de um pacote atrasado dentro da rede. Num canal em que os pacotes são entregues por ordem, é necessário garantir que o pacote anterior não seja confundido com o que acabou de ser emitido. Por agora vamos admitir que os números de sequência usados têm o número suficiente de bits para garantir que essas condições são satisfeitas e portanto a associação entre pacotes e números de sequência é inequívoca.

Nos protocolos apresentados a seguir, por hipótese, é sempre possível fazer progresso, *i.e.*, o fluxo de dados nunca bloqueia num dado pacote que nunca é entregue ao destino com sucesso. Na prática, podem existir situações em que tal não é possí-

vel pelas mais diversas razões. Essas situações são detectadas impondo um limite ao número máximo de vezes que um pacote é retransmitido. Quando esse número é ultrapassado, o emissor desiste de tentar transmitir qualquer informação para o receptor. Esta condição excepcional de terminação não é explicitamente apresentada.

6.2 Protocolo *stop & wait*

O protocolo mais simples que assegura fiabilidade com retransmissão é designado na literatura de redes por protocolo *stop & wait*, e tem uma filosofia relativamente simples: sempre que é emitido um pacote de dados, só se pode passar ao seguinte depois de receber a confirmação da sua recepção. Caso essa confirmação não chegue, ou chegue uma indicação de que o pacote está em falta, reemite-se o mesmo pacote. O protocolo chama-se assim pois progride normalmente por um ciclo composto pela emissão de um pacote de dados, seguida de espera pela confirmação da recepção (ACK) do mesmo.

O protocolo *stop & wait* é um protocolo em que o emissor envia um pacote e só passa a enviar o pacote seguinte quando receber a confirmação da sua recepção (ACK) vinda do receptor.

A Listagem 6.1 apresenta a parte cíclica da actividade do emissor e do receptor durante a transferência dos dados.

Correcção do protocolo *stop & wait*

O protocolo é muito simples e por isso é relativamente fácil discutir a correcção do mesmo. Por um lado, o emissor repete a emissão do mesmo pacote até receber a confirmação de que o último pacote emitido foi recebido pelo receptor. Isso garante a condição: um pacote só se considera processado pelo emissor depois de este ter a certeza de que o receptor o recebeu. Por outro lado, a existência do temporizador e a ocorrência de *timeouts* quando não há recepção atempada garantem que o emissor não fica bloqueado e vai continuar a enviar o pacote corrente. Qualquer evento inesperado é ignorado. Como, por hipótese, mais tarde ou mais cedo os pacotes serão entregues, isso garante que o emissor não entra em ciclo eterno desde que o receptor retransmita os ACKs do último pacote bem recebido.

O receptor só guarda um pacote se o mesmo corresponder ao pacote que está à espera. Se tal for o caso, envia o ACK respectivo e passa a esperar pelo pacote seguinte. Senão, envia um ACK do último pacote bem recebido e continua à espera do mesmo pacote. Portanto, o receptor só confirma a recepção de um pacote quando este foi de facto recebido. O envio do ACK repetido é necessário para compensar a eventual perda do ACK mais recente. Como mais tarde ou mais cedo os dados acabarão por chegar ao receptor e os ACKs ao emissor, mesmo que seja necessária a sua retransmissão, é fácil aceitar que o protocolo vai fazendo progresso.

Ficam assim respondidas questões como: o protocolo resiste à perda de pacotes ou à sua chegada fora de ordem? A resposta é sim. Garante-se também que, mesmo nas condições mais adversas, existe uma relação inequívoca entre os números de sequência e os pacotes. De facto, por hipótese, os números de sequência têm um número de bits que garante que um número de sequência só é reutilizado depois de se garantir que o pacote que o utilizou mais recentemente já não existe.

Uma última questão diz respeito ao valor do *timeout*. Qual deve ser o seu valor? Que acontece se este está mal calculado? O valor do *timeout* tem de acomodar o RTT (tempo de trânsito entre o emissor e o receptor e vice-versa), assim como o tempo de processamento pelo receptor, que pode estar mais ou menos carregado com outras

Listing 6.1: Pseudo código da parte cíclica do protocolo *stop & wait* sem limitação do número de retransmissões.

```

Sender:
    seq = 1
    connection.create()

    while ( more application packets to send ) {
        packet = getNextPacketToSend()
        packet.opCode = DATA
        packet.sequence = seq
        // loop until the packet is acked
        while ( true ) {
            send(packet);
            startTimer( duration )

            On event < ACK received and ACK.sequence() == seq >:
                stopTimer()
                seq++
                break // leave loop

            On event < TIMEOUT >:
                // loop again
        }
        connection.close()
    }

Receiver:
    seq = 1

    while ( connection.isActive() ) {

        On event < DATA packet received and packet.sequence == seq >:
            // received the expected packet
            deliverPacketToApplication(packet)
            ACK.sequence = seq
            send(ACK)
            seq++

        On event < DATA packet received and packet.sequence != seq >:
            // ignore packet and resend the previous ACK
            ACK.sequence = seq - 1
            send(ACK)
    }
}

```

tarefas. Se o *timeout* for demasiado longo, a recuperação de erros será mais lenta mas a correcção do protocolo não será colocada em causa. Se o *timeout* for demasiado curto, serão retransmitidos pacotes inutilmente mas a correcção do protocolo também não será violada.

Acima foram introduzidos os pacotes NACK. No protocolo *stop & wait* um NACK pode ser enviado quando for recebido um pacote com erros. No entanto, nessa situação tanto um NACK como um ACK do último pacote bem recebido desarmam o temporizador no emissor e desencadeiam uma retransmissão, pelo que ambas as soluções têm o mesmo efeito.

Como já referimos, esta discussão ignora os problemas de inicializar e terminar a transferência. Um exemplo de como essas questões podem ser abordadas será apresentado aquando da discussão do protocolo TCP, ver o Capítulo 7.

Dado parecer claro que estamos perante um protocolo simples que cumpre o objectivo, o próximo passo consiste em avaliar o desempenho do mesmo.

Desempenho do protocolo *stop & wait* sem erros

O melhor desempenho do protocolo coincide com o seu funcionamento sem erros, pois a recuperação dos erros introduz desperdícios que prejudicam o desempenho. Como, no caso geral, o tempo de trânsito dos pacotes não é constante e o tempo de processamento pelo receptor também não, é habitual fazer as seguintes simplificações: 1) admite-se que o RTT médio é estável e toma-se o seu valor como uma constante; quando entre o emissor e receptor está um único canal, admite-se que RTT é igual a duas vezes o tempo de propagação; 2) admite-se que o receptor está bem dimensionado e despreza-se o tempo de processamento do mesmo; 3) admite-se que o tempo de transmissão dos pacotes de dados é constante pois os mesmos têm dimensão constante e suficientemente grande para se desprezar o tempo de transmissão dos cabeçalhos; e finalmente, 4) admite-se que os pacotes de ACK só têm cabeçalho e portanto despreza-se o seu tempo de transmissão.

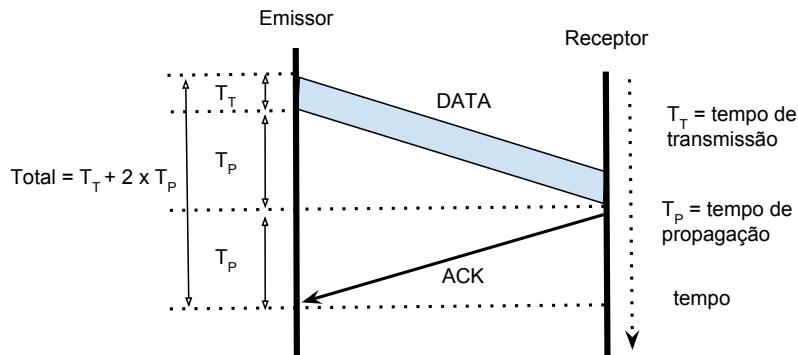


Figura 6.9: Determinação do desempenho do protocolo *stop & wait*

Com estas hipóteses, ver a Figura 6.9, o tratamento de cada pacote é repetido ciclicamente, cada iteração tem por duração $T_{total} = T_T + 2.T_P$ e durante cada uma transmite-se a quantidade de bits correspondentes a um pacote. Portanto, o débito médio extremo a extremo (V) será a dimensão em bits dos pacotes de dados (D) a

dividir por T_{total} .

$$V = \frac{D}{T_T + 2.T_P} = \frac{D}{T_T + RTT} \quad (6.1)$$

Quando o protocolo é aplicado num contexto em que existe um único canal *full-duplex* entre o emissor e o receptor, define-se a **taxa de utilização** (*usage ratio*) do canal como sendo a fracção do tempo em que o canal se encontra a transmitir dados do emissor para o receptor. Denotando a taxa de utilização por T_U , a mesma é dada por

$$T_U = \frac{T_T}{T_T + 2.T_P} = \frac{T_T}{T_T + RTT} \quad (6.2)$$

Olhando para a Figura 6.9 é fácil reconhecer a validade da equação 6.2 e ainda como a relação entre o RTT e o T_T a influenciam. Se o valor do RTT é desprezável face ao T_T , a taxa tende para 100%, se o valor do RTT é muito maior que o T_T , a taxa aproxima-se de 0.

Assim, sempre que estamos numa situação em que o T_P é desprezável, a taxa de utilização aproxima-se de 1 e o débito extremo a extremo (V) aproxima-se do débito do canal. Sempre que o RTT é significativo e vai crescendo, a taxa tende para 0, e isso que dizer que o débito disponível do canal é cada vez menos utilizado.

Por exemplo, admitindo que os pacotes têm 10^4 bits e que o débito do canal (V_T - Velocidade de Transmissão) é 1 Mbps, $T_T = 10^4/10^6 = 10^{-2} = 10\ ms$. Admitindo adicionalmente que o RTT é de 0,1 ms, $T_U = 10/(10 + 0.1) = 99\%$. A Tabela 6.1 a seguir mostra as diversas taxas de utilização conseguidas em diversos contextos.

Tabela 6.1: Taxas de utilização obtidas pelo protocolo *stop & wait* com pacotes de dados de 10.000 bits, canal sem erros, desprezando o tempo de transmissão dos ACKs e variando o débito (V_T) e o RTT do canal

T_U	V_T (Mbps)	T_T (ms)	RTT (ms)	Caracterização
99%	1	10	0,1	V_T baixa, RTT desprezável
50%	100	0,1	0,1	V_T razoável, RTT desprezável
33%	1	10	20	V_T baixa, RTT baixo
0,5%	100	0,1	20	V_T razoável, RTT baixo
4,8%	1	10	200	V_T baixa, RTT alto
0,05%	100	0,1	200	V_T razoável, RTT alto
0,005%	1000	0,01	200	V_T alta, RTT alto

Os valores de T_U na tabela confirmam o que é evidente. O protocolo *stop & wait* não transfere mais do que um pacote por RTT. Perante um elevado RTT, caso ilustrado na parte direita da Figura 6.10, a taxa de utilização, assim como o débito médio, são muito baixos. Para se aumentar a taxa de utilização, é necessário arriscar e enviar mais pacotes, *i.e.*, enviar pacotes de avanço mesmo que não se tenha ainda a certeza de que alguns dos anteriores foram bem recebidos, como está ilustrado na parte esquerda da mesma figura.

6.3 Protocolos de janela deslizante

Existe um conjunto de protocolos do tipo ARQ, *i.e.*, com base em retransmissão dos pacotes pelo emissor, que se designam por protocolos de janela deslizante ou *pipelining*

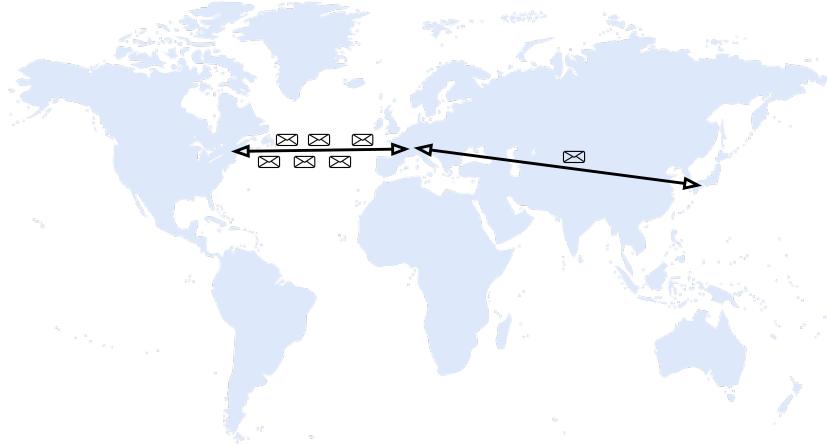


Figura 6.10: À direita dois interlocutores usam o protocolo *stop & wait*, que envia no máximo um pacote por RTT; à esquerda dois interlocutores usam um protocolo que envia mais do que um pacote por RTT

protocols. A ideia comum a estes protocolos consiste em ir enviando pacotes para a frente, mesmo sem receber ACKs dos pacotes emitidos anteriormente, na esperança de que os erros não sejam muito frequentes e com isso se aumente a velocidade de transferência extremo a extremo.

Os pacotes que já foram emitidos mas de que o emissor ainda não recebeu a confirmação de recepção, *i.e.*, que não foram ainda “acked”, dizem-se pacotes em trânsito (*in-transit or in-flight packets*). Todos estes protocolos limitam o número máximo de pacotes em trânsito em cada momento (N) por diversas razões. Primeiro por razões de controlo de fluxo, *i.e.*, de forma a que um emissor muito rápido não “afogue” um receptor lento a processar os pacotes. Outra razão tem a ver com a necessidade de não enviar mais pacotes do que aqueles que a rede consegue, no momento, encaminhar entre o emissor e o receptor sem demasiadas supressões de pacotes. Enviar pacotes a um ritmo superior ao que a rede aguenta diminui o desempenho e desperdiça recursos. Finalmente, pode ser inútil usar um valor de N demasiado grande, dependendo da taxa de erros da rede, porque o custo de corrigir um erro pode implicar a retransmissão de pacotes já em trânsito.

A Figura 6.11 mostra o caso em que $N = 3$, ou seja, o número de pacotes *in-flight* é no máximo 3. Se não ocorrerem erros, em cada ciclo de emissão de pacotes (que continua a durar $T_{total} = T_T + 2.T_P$), ao invés de se transferir um pacote para o receptor, foram transferidos $N = 3$ pacotes. O débito extremo a extremo aumentou N vezes.

Como ainda não foi confirmada a recepção dos pacotes emitidos, é necessário que o emissor mantenha uma cópia dos mesmos para os poder vir a retransmitir se necessário. Assim, o emissor tem de dispor de um *buffer* de emissão com espaço para N pacotes, o qual se designa **por janela de emissão de pacotes**. Portanto, a janela de emissão de pacotes é um *buffer* de dimensão N que contém os pacotes em trânsito. Que acontece quando a janela está cheia, *i.e.*, quando existem N pacotes em trânsito? Mais nenhum pacote pode ser emitido até chegarem ACKs dos emitidos. Se tudo estiver a correr bem, o número de sequência do primeiro ACK a chegar será o do pacote mais antigo na janela, que assim abandona a janela e esta diminui de uma unidade. Portanto, um

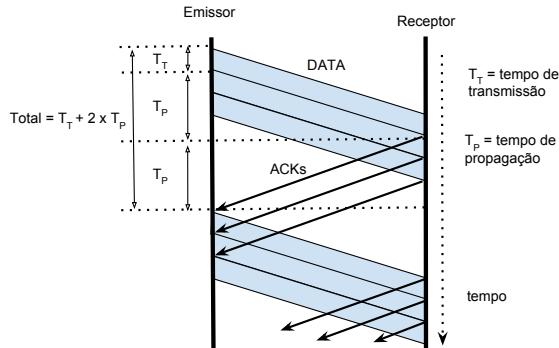


Figura 6.11: Aumento da velocidade de transferência extremo pelo aumento do número de pacotes *in-flight*

novo pacote pode ser emitido sem violar a condição: a dimensão da janela de emissão é $\leq N$, e assim sucessivamente.

Protocolos de janela deslizante (*sliding window protocols*) são protocolos em que o emissor envia vários pacotes de avanço, mesmo sem receber os ACKs referentes aos mais antigos. Os pacotes em trânsito são mantidos num *buffer* de emissão chamado a janela do emissor. A dimensão desta janela é limitada por razões de controlo de fluxo e de controlo da saturação da rede. Adicionalmente, a sua dimensão pode também ser limitada para se evitar desperdício em caso de perda de pacotes.

Torna-se agora mais claro porque este *buffer* do emissor se chama janela do emissor. Considerando os números de sequência do fluxo de pacotes que o emissor quer enviar ao receptor, podemos assimilar o *buffer* a uma janela sobre esse fluxo, ver a Figura 6.12, que contém um subconjunto contíguo de números de sequência (pacotes), correspondente aos pacotes em trânsito, ou que podem ser emitidos mantendo a condição dimensão da janela $\leq N$. Os números de sequência à esquerda da janela correspondem a pacotes de que o receptor já recebeu a confirmação de recepção (já foram “acked”). À direita da janela temos os números de sequência dos pacotes que irão ser transmitidos no “futuro”.

Qual a dimensão do *buffer* do receptor? A solução mais simples consiste em o receptor só ter espaço para um pacote. Quer isto dizer que se o pacote que o receptor receber não for o exactamente o pacote de que ele estava à espera, o mesmo é desprezado. Esta primeira alternativa corresponde a um protocolo designado GBN.

Protocolo voltar atrás N (GBN - Go-back-N)

O protocolo GBN é um protocolo de janela deslizante em que a janela do emissor tem dimensão N e a janela do receptor tem (pelo menos) dimensão 1. O emissor, desde que tenha pacotes para enviar, e enquanto a condição sobre a dimensão da janela de emissão ser $\leq N$ for respeitada, emite pacotes uns atrás dos outros. O receptor, desde que receba um pacote com o número de sequência de que está à espera, aceita-o, processa-o (e.g., entrega-o à aplicação) e envia um ACK, ver a Figura 6.13.

No entanto, se o receptor receber um pacote com o número de sequência errado (i.e., um duplicado ou um pacote adiantado porque foi alterada a ordem de entrega dos

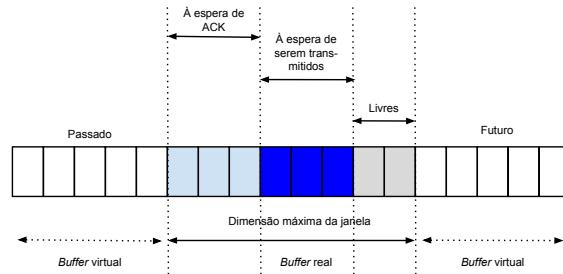


Figura 6.12: Janela do emissor nos protocolos de janela deslizante

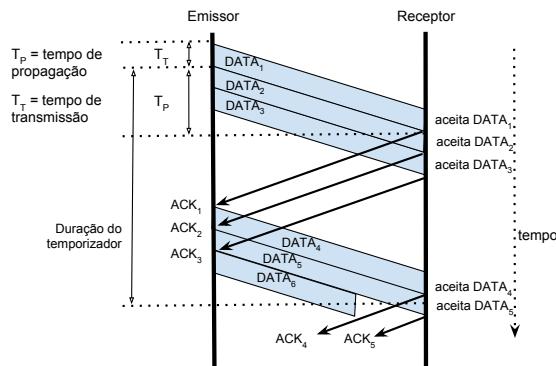


Figura 6.13: O protocolo GBN evoluindo sem erros

pacotes, ou o anterior perdeu-se) despreza-o, e retransmite um ACK correspondente ao último pacote que recebeu correctamente, ver a Figura 6.14.

Quando dispara um alarme no emissor (*timeout*), este volta a reemitir os pacotes que estão na janela, começando pelo que tem o número de sequência mais baixo, pois este é o pacote em trânsito mais antigo, e para poder voltar a colocar os pacotes pela ordem é necessário recomeçar por este e reemitir a janela. É por esta razão que o protocolo se chama GBN, porque em caso de anomalia, o emissor regressa ao pacote mais antigo (*i.e.*, recua N pacotes) e recomeça a emissão da janela.

As acções executadas pelo emissor do protocolo GBN são apresentadas a seguir na Listagem 6.2. O receptor é idêntico ao receptor do protocolo *stop & wait*.

A janela é uma fila de pacotes em que os novos pacotes a transmitir são acrescentados no fim, e o primeiro pacote é o pacote em trânsito mais antigo. O *timeout* é colocado quando é enviado o primeiro pacote da janela. Sempre que se recebe um ACK do pacote mais antigo na janela, *i.e.*, na primeira posição da fila, este deixa a janela e o valor do temporizador é ajustado. Se a janela ficou vazia, pára-se o temporizador, senão inicializa-se o temporizador com o valor *duration* – *t* em que *t* corresponde ao tempo que já decorreu desde que o novo pacote em trânsito mais antigo foi emitido. O procedimento *GoBackN* consiste em posicionar o pacote mais antigo da janela como o próximo pacote a emitir.

É possível introduzir uma optimização aquando da recepção de um ACK com

Listing 6.2: Pseudo código da parte cíclica do emissor no protocolo GBN com receptor com janela de recepção de dimensão 1 e sem limitação do número de retransmissões.

```
// "window" is a queue with maximum length N and
// with item positions from 0 to window.size()-1 if not empty
seq = 1
connection.create()

while ( more application packets to send or window.size() > 0 ) {

    On event < end of last packet transmission and window.size() == 0 >:
        // ignore it and loop again

    On event < end of last packet transmission and window.size() != 0 >:
        position = window.getNextPacketPosition() // next packet to send
        window.sendPacket(position)
        if ( position == 0 ) startTimer(duration)

    On event < ACK and ACK.sequence == window.getSequenceNumber(0) >:
        window.removeFirst()
        adjustTimer( window.getExpirationTime(0) )

    On event < ACK and ACK.sequence != window.getSequenceNumber(0) >:
        // ignore it and loop again

    On event < applicationPacketAvailable and window.size() < N >:
        packet = getNextApplicationPacket()
        packet.opCode = DATA
        packet.sequence = seq
        window.addLast(packet)
        seq++

    On event < TIMEOUT >:
        // The next packet to transmit is the one at position 0 of window
        window.GoBackN()
        next event = end of last packet transmission
}

connection.close()
```

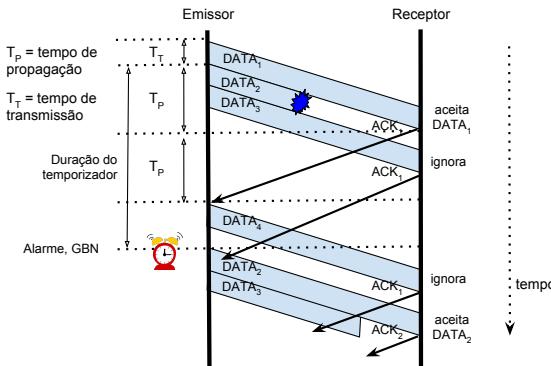


Figura 6.14: Ilustração do mecanismo GBN na sequência do disparo de um alarme

número de sequência repetido. A recepção de um ACK com o número de sequência repetido, para além de permitir compensar a perda de um ACK, permite também ao emissor aperceber-se mais cedo que houve uma anomalia e o pacote se perdeu sem necessitar de esperar pelo eventual disparo do *timeout*. Ver a Figura 6.15.

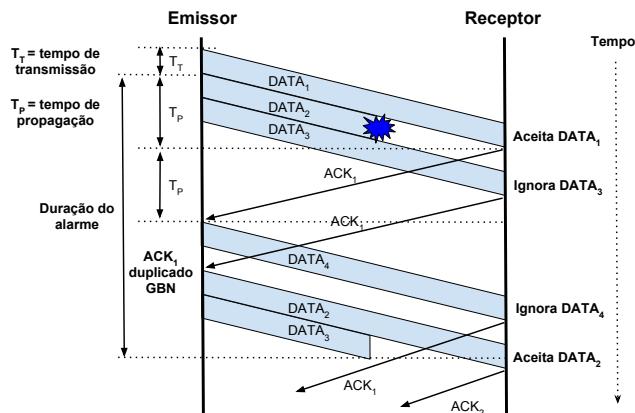


Figura 6.15: Ilustração da optimização possível na sequência de um ACK duplicado

A Figura 6.12 mostra a janela do emissor durante a evolução do protocolo GBN e a Figura 6.13 a evolução normal do protocolo sem erros.

O protocolo GBN (*Go-back-N*) é um protocolo de janela deslizante que usa uma janela de emissão maior que 1. Quando dispara um alarme (*timeout*), o emissor recomeça a enviar todos os pacotes que ainda estão em trânsito, a começar pelo mais antigo.

Protocolo Go-back-N com janela do receptor maior que 1

É relativamente fácil de imaginar uma forma de melhorar o protocolo GBN através do aumento da dimensão da janela do receptor. Nessa realização do protocolo os números de sequência dos ACKs são cumulativos, *i.e.*, um ACK com o número de sequência n significa que todos os pacotes até ao de número de sequência n foram correctamente recebidos.

Assim, quando um pacote chega fora de ordem, o receptor envia o ACK com o número de sequência do último pacote bem recebido, mas guarda o pacote se tiver lugar na janela de recepção. Assim que chegar um pacote que colmate a falta, o novo ACK, como é cumulativo, considera todos os pacotes correctamente recebidos até aí.

Por exemplo, na Figura 6.16 quando chega o pacote 1, o receptor envia um ACK 1, depois chegam os pacotes 3 e 4 e o receptor guarda-os mas envia sempre um NACK. O emissor quando recebe o NACK, decide iniciar imediatamente o processo GBN e volta a reemitir o pacote 2. Finalmente, quando o receptor recebe o pacote 2, envia um ACK 4 pois até ao momento já recebeu correctamente os pacotes 1, 2, 3 e 4.

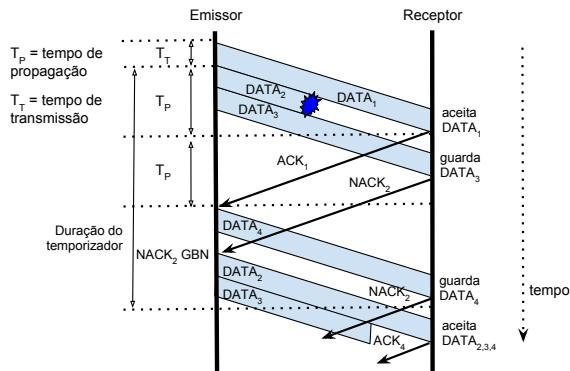


Figura 6.16: Protocolo GBN com o receptor com janela maior que 1 e a guardar os pacotes recebidos fora de ordem

A utilização de um ACK cumulativo, a utilização de uma janela do receptor maior que 1 e o desencadeamento do processo GBN sempre que é recebido um NACK ou ACKs duplicados, permite uma recuperação potencialmente mais rápida dos erros ocorridos.

Constata-se que nos casos em que o RTT é bastante significativo e a capacidade dos canais elevada, é necessário usar janelas de dimensão bastante elevadas, para que seja possível explorar adequadamente o débito extremo a extremo permitido pela rede. Nestes casos, a perda de um pacote, dado o elevado RTT e a dimensão da janela, só será recuperada depois de terem sido transmitidos muitos pacotes para a frente. O funcionamento Go-Back-N leva então à retransmissão inútil de muitos pacotes como mostra a Figura 6.17.

Existe uma outra versão do protocolo de janela deslizante em que a janela do receptor também é maior que 1, mas que permite recuperar mais rapidamente da perda de pacotes e tenta evitar o desperdício devido a retransmissões inúteis. Essa versão é designada repetição selectiva.

Antes de a apresentarmos, convém chamar a atenção para o facto de que se entre o emissor e o receptor está um único canal, este não troca a ordem dos pacotes, a

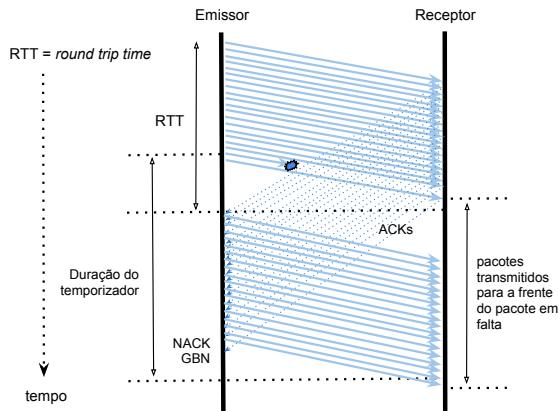


Figura 6.17: Quando a janela é muito grande, o funcionamento GBN leva à potencial retransmissão inútil de muitos pacotes que já foram transmitidos e recebidos para a frente do pacote em falta

recepção de um ACK cumulativo repetido (ou um NACK) é sinónimo da perda de um pacote e o seu reenvio imediato justifica-se. No caso de uma rede que pode trocar a ordem dos pacotes, a chegada de ACKs cumulativos (ou de NACKs) não deve ser interpretada imediatamente como equivalente a um evento *timeout*. Por exemplo, o protocolo TCP usa este tipo de mecanismos e só ao fim da recepção de 3 ACKs repetidos é que desencadeia a nova emissão.

Protocolo repetição selectiva (*Selective Repeat – SR*)

O protocolo de repetição selectiva (*Selective Repeat – SR*) usa janelas de dimensão maior que 1 no emissor e no receptor. No entanto, a diferença para o GBN está no processamento dos ACKs, pois estes deixam de ser cumulativos e passam a dizer respeito apenas a um único pacote. Um ACK do pacote *seq* significa apenas que o pacote *seq* foi bem recebido, não acrescentando nenhuma informação sobre outros pacotes. O mecanismo GBN deixa de ser usado, visto que o emissor nunca recua N pacotes. O emissor associa um temporizador separado a cada pacote emitido. Quando o respectivo alarme dispara, apenas o pacote ao qual está associado é retransmitido e retoma-se o processamento normal dos outros pacotes após essa retransmissão.

O objectivo do protocolo SR é evitar, tanto quanto possível, a retransmissão de pacotes que tenham sido bem recebidos. Assim, só são retransmitidos os pacotes para os quais disparou o *timeout*, ou foi recebido um NACK, e mais nenhum outro. Os outros pacotes, caso sejam recebidos os respectivos ACKs antes de o respectivo alarme disparar, não serão retransmitidos.

Existe apenas uma excepção ao caso da retransmissão, que é quando o valor do temporizador se revela curto para um dado pacote. Neste caso pode ocorrer um alarme prematuro e ser retransmitido um pacote ainda em trânsito.

Para acelerar o protocolo, sempre que o receptor recebe um pacote que cabe na janela mas está fora de sequência e abre um “buraco” na sequência de pacotes porque existem um ou mais pacotes em falta, para além de enviar um ACK do pacote corretamente recebido, o receptor pode enviar um NACK do pacote em falta, na esperança de se adiantar ao temporizador associado ao pacote no emissor e levá-lo a retransmitir o pacote tão cedo quanto possível.

O protocolo SR (*Selective Repeat protocol*) é um protocolo que usa janelas de emissão e recepção maiores que 1. O protocolo trata cada pacote em trânsito de forma independente. Quando dispara um alarme (*timeout*), o emissor apenas reenvia o pacote ao qual o alarme está associado.

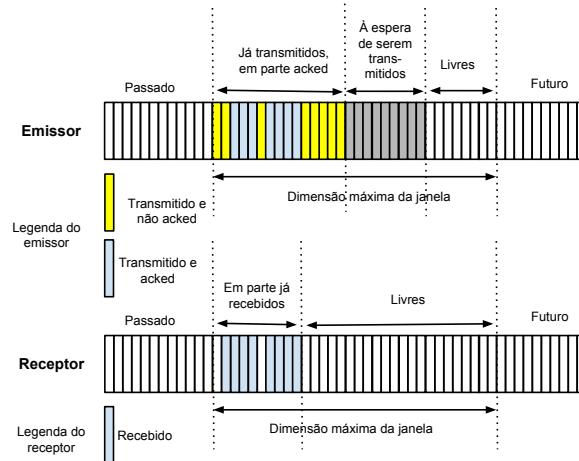


Figura 6.18: Janelas do emissor e do receptor no protocolo SR

A Figura 6.18 mostra as janelas do emissor e do receptor durante a evolução do protocolo. Em ambas as janelas, a condição de a dimensão ser $\leq N$ tem de ser respeitada. Assim, se a janela do emissor estiver cheia, este só pode aceitar mais pacotes da aplicação para transmitir quando receber o ACK do pacote em trânsito mais antigo. Por outro lado, o receptor só pode entregar à aplicação dados sem “buracos”. Se o receptor tem a janela cheia, mas com falta de pacotes “à esquerda”, pode optar por transmitir NACKs dos mesmos, para tentar desencadear a sua retransmissão tão cedo quanto possível.

A Figura 6.19 mostra um exemplo do funcionamento do protocolo que permite ver que de facto o protocolo evita retransmissões inúteis. Quando for recebido o ACK do pacote mais antigo, a janela pode deslocar-se N pacotes para a direita, visto que os pacotes seguintes estão todos *acked*. É, no entanto, necessário ter em atenção que o receptor pode receber pacotes抗igos repetidos fora da janela, porque se perderam os respetivos ACKs. Nesse caso o receptor terá também de enviar um novo ACK dos mesmos.

O protocolo de repetição selectiva é mais complexo no seu conjunto, sobretudo no que diz respeito aos temporizadores, razão pela qual nem sempre é adoptado. Resumidamente, o seu funcionamento é o seguinte:

Apesar de o protocolo SR ser mais complexo que os protocolos anteriores, sobretudo no que diz respeito à gestão dos alarmes, é relativamente fácil convencermos-nos da sua correção pois cada pacote é tratado de forma isolada. Cada pacote particular só é considerado transferido pelo emissor quando este recebe o ACK respectivo. Dada a utilização de alarmes associados a cada pacote, enquanto o ACK não for recebido, esse pacote é retransmitido. Por outro lado, se algum ACK se perder, o protocolo assegura que serão enviadas cópias dos ACKs quando o receptor receber duplicados.

Listing 6.3: Pseudo código da parte cíclica do emissor do protocolo SR sem limitação do número de retransmissões.

```
// "window" is a queue with maximum length N and
// with item positions from 0 to window.size()-1 if not empty
seq = 1
connection.create()

while ( more application packets to send or window.size() > 0 ) {

    On event < end of last packet transmission and window.size() == 0 >:
        // ignore it and loop again

    On event < end of last packet transmission and window.size() != 0 >:
        position = window.getNextPacketToSend()
        window.sendPacket(position)
        // Start a timer for this packet
        startTimer( duration, window.getSequenceNumber(position) )

    On event < ACK and ACK.sequence in window >:
        window.markAsAcked(ACK.sequence)
        deleteTimer(ACK.sequence)
        // Remove acknowledged packets at the start of window if any
        while (window.size() > 0 && window.getFirst().isAcked())
            window.removeFirst()

    On event < ACK and ACK.sequence not in window >:
        // ignore it and loop again

    On event < TIMEOUT or NACK >: // both events have a sequence number
        position = window.getPacketPosition(event.sequence)
        if (the packet is still in the window) {
            window.sendPacket(position)
            startTimer( duration, window.getSequenceNumber(position) )
        }

    On event < applicationPacketAvailable and window.size() < N >:
        packet = getNextApplicationPacket()
        packet.opCode = DATA
        packet.sequence = seq
        window.addLast(packet)
        seq++
}

connection.close()
```

Listing 6.4: Pseudo código da parte cíclica do receptor do protocolo SR

```
seq = 1
while ( connection.isActive() ) {

    On event < DATA and DATA.sequence in window >:
        send(ACK, DATA.sequence);
        position = window.getPacketPosition(DATA.sequence);
        window.put(DATA, position);
        if (position != 0) send(NACK, window.getFirst().sequence())
        // Try to deliver as many packets as possible to the application
        while (window.size() > 0 and window.getFirst().isAvailable()) {
            window.deliverFirstToApplication();
            window.deleteFirst();
        }

    On event < DATA and DATA.sequence not in window >:
        send(ACK, DATA.sequence); // resend ack
}
```

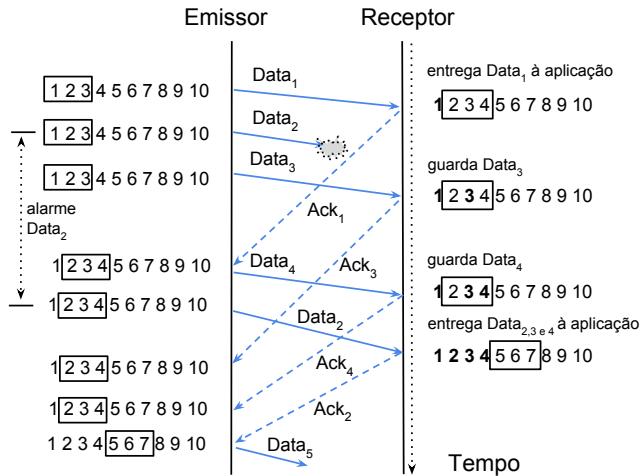


Figura 6.19: O protocolo SR em ação, com recuperação mais rápida do erro e sem retransmissão de pacotes já recebidos pelo receptor

Como por hipótese, mais tarde ou mais cedo, os pacotes enviados do emissor para o receptor, e vice versa, acabam por chegar ao destino, o protocolo acabará por fazer progresso. Adicionalmente, por hipótese, existe uma associação inequívoca entre números de sequência e pacotes, pelo que não pode resultar alguma confusão entre os números de sequência dos pacotes e dos respectivos ACKs.

Finalmente, as janelas só avançam de forma que as condições anteriores se verificam. A janela do emissor só avança quando o emissor tem a certeza de que o receptor recebeu os pacotes. A janela do receptor só avança quando contém pacotes em sequência e que portanto podem ser entregues à aplicação sem violar as condições de fiabilidade do protocolo.

Tal como já vimos a propósito do protocolo *stop & wait*, a regulação da duração do alarme é um problema delicado com impacto apenas sobre o desempenho, mas sem implicações na correcção. Uma duração de *timeout* demasiado baixa provoca a emissão prematura de pacotes ainda em trânsito. Um valor de *timeout* demasiado elevado, atrasa a recuperação de erros. Por outro lado, a utilização de NACKs apenas serve para levar o emissor a retransmitir um pacote em falta o mais cedo que possível, mesmo sem a duração do respectivo *timeout* se esgotar completamente.

Comparação entre SR e GBN

Um dos aspectos mais complexos da implementação do protocolo SR tem a ver com a gestão dos *timeouts*. Naturalmente, a existência de tantos alarmes activos quantos os pacotes na janela (a janela de emissão pode em certas circunstâncias necessitar de conter várias centenas de pacotes) seria complexo e pouco realista. No entanto, é possível associar a cada pacote na janela a hora a que este deve ser reemitido se no entretanto não tiver sido *ack'd*, e usar apenas um alarme. Quando este dispara, é necessário determinar que pacote(s) já deveriam ter sido *ack'd(s)*, retransmitir esse(s) pacote(s), calcular de novo quanto tempo falta para a hora de retransmissão mais próxima entre os pacotes presentes na janela, e activar de novo apenas um alarme com esse valor.

De resto os dois protocolos só diferem na forma como tratam os erros e desenca-deiam as retransmissões. O protocolo SR tem tendência a transmitir menos duplicados

que o GBN na medida em que o SR só retransmite inutilmente um pacote quando o respectivo ACK se perdeu, enquanto que o GBN tem tendência a retransmitir mais duplicados, pois ao voltar atrás recomeça a transmissão de todos os pacotes que se seguem ao que foi responsável pelo *timeout*, mas alguns desses podem já ter chegado. Sobretudo com grandes janelas, uma perda esporádica de um pacote implica a transmissão de menos pacotes em duplicado com SR do que com GBN. No entanto, sem NACKs, o GBN ao detectar ACKs cumulativos repetidos recupera das perdas mais depressa pois não tem de esperar pelo desencadear do alarme (*timeout*), que é o único meio à disposição do SR para detectar perdas nesse cenário.

Provavelmente, como veremos mais tarde, uma solução mais interessante passaria por associar as vantagens dos dois protocolos, *i.e.*, juntar os ACKs cumulativos do GBN com um mecanismo de sinalização do que foi e não foi recebido correctamente para a frente do ACK cumulativo. Esta é solução adoptada em opção pelo protocolo TCP, ver a Secção 7.2.

De qualquer forma, uma observação atenta dos diferentes protocolos de janela deslizante permite concluir que estes apenas diferem no método de recuperação dos erros. Se não ocorrerem erros, a velocidade de transferência de dados do emissor para o receptor só depende da dimensão da janela do emissor e dos mesmos parâmetros que o protocolo *stop & wait*. Por esta razão torna-se fácil analisar o desempenho dos protocolos de janela deslizante num cenário em que não ocorram erros.

Desempenho do protocolo de janela deslizante sem erros

A Figura 6.20 mostra o progresso de uma transferência do emissor para o receptor quando o emissor tem uma janela de emissão de dimensão $N > 1$.

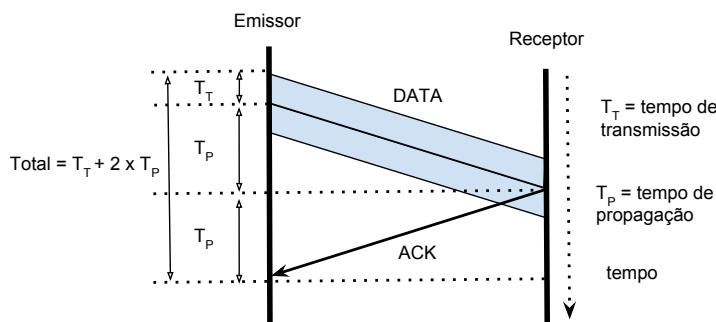


Figura 6.20: Progresso de um protocolo de janela deslizante sem erros e com $N = 2$

Com as mesmas hipóteses que as usadas aquando da avaliação do desempenho do protocolo *stop & wait*, cada ciclo tem uma duração $T_{total} = T_T + 2.T_P$ e durante o mesmo transmite-se a quantidade de bits correspondentes aos pacotes da janela. Portanto, o débito extremo a extremo (V) será a dimensão em bits dos N pacotes de dados (D) a dividir por T_{total} quando $N.T_T < T_T + 2.T_P$

$$V = \frac{N.D}{T_T + 2.T_P} = N.V_{stop \& wait} \quad (6.3)$$

ou a velocidade máxima do emissor quando a janela não se esgota em $T_T + 2.T_P$. A

taxa de utilização (*usage ratio*), nas mesmas condições, é dada por

$$T_U = \frac{N \cdot T_T}{T_T + 2 \cdot T_P} = N \cdot T_{U \text{stop \& wait}} \quad (6.4)$$

ou 100% quando a janela também não se esgota em $T_T + 2 \cdot T_P$.

Nas situações em que o tempo de transmissão seja desprezável face ao RTT ($T_T \approx 0$ e $RTT \approx 2 \cdot T_P$), a velocidade de transmissão extremo a extremo, se não ocorrerem erros, aproxima-se do número de bits que cabem na janela, a dividir pelo RTT

$$V = \frac{N \cdot D}{RTT} \quad (6.5)$$

Sem ocorrência de erros, os desempenhos dos protocolos GBN e SR são iguais vistos que ambos transferem ao ritmo máximo permitido pelo protocolo de janela deslizante. No entanto, a recuperação de erros é geralmente mais lenta no GBN como já se discutiu.

A ocorrência de erros implica que a transferência de alguns pacotes dura mais do que $T_T + 2 \cdot T_P$, com eventual impacto no atraso da transferência de outros pacotes, sobretudo no caso do GBN. Dado que a ocorrência de erros tem um carácter estocástico, a dedução do desempenho em função da distribuição dos erros é mais elaborada, ver as referências na secção 6.5.

Nomenclatura e parâmetros dos protocolos de janela deslizante

Todos os protocolos vistos neste capítulo são caracterizáveis, no essencial, pelo tamanho das janelas que usam. Na verdade, o protocolo *stop & wait* é um protocolo de janela deslizante com as janelas de emissão e recepção iguais a 1. A tabela a seguir apresenta essa visão.

Tabela 6.2: Dimensão das janelas usadas pelos protocolos de janela deslizante

Protocolo	Emissor	Receptor
<i>Stop & Wait</i>	1	1
<i>Go-Back-N</i>	N	1
<i>Selective Repeat</i>	N	N

Os protocolos de janela deslizante podem ser usados ao nível dos canais, ou aos níveis transporte ou aplicação. Quando os canais dominantes tinham uma taxa de erros significativa, eram usados protocolos deste tipo entre as extremidades de cada canal para compensar imediatamente os erros detectados. Hoje em dia, a maioria dos canais exibem taxas de erro muito baixas, especialmente os baseados em fibra óptica, e assim deixa-se aos níveis transporte ou aplicação a responsabilidade de recuperar os erros extremo a extremo. A excepção a esta regra são os canais sem fios, que continuam a exhibir taxas de erros significativas. Esperar pela recuperação extremo a extremo depende de um valor de *timeout* necessariamente mais elevado que o RTT do canal, e até que um pacote seja retransmitido, o canal esteve subaproveitado.

Por essa razão, quase todos os canais sem fios, em particular os canais Wi-Fi, usam um protocolo de retransmissão de pacotes ao nível canal. Os canais Wi-Fi transmitem no máximo a uma centena de metros, como a velocidade de propagação do sinal no ar é cerca de 300.000.000 m/s, o tempo de propagação máximo é de $0,33 \cdot 10^{-6}$ segundos ou $0,33 \mu\text{s}$. Por esta razão esses canais usam o protocolo *stop & wait*.

No início desta secção já discutimos os constrangimentos a que tem de obedecer a escolha da dimensão da janela do emissor. Em protocolos usados directamente em canais essa dimensão é fixa. No entanto, nos protocolos de janela deslizante usados nos

níveis superiores essa dimensão é geralmente variável. A mesma depende do estado da rede e da capacidade do receptor e das aplicações consumirem atempadamente os pacotes novos chegados, *i.e.*, por razões de controlo da saturação da rede, ver o Capítulo 8, e de controlo de fluxo.

Como também referimos atrás, os números de sequência, dado darem a volta, precisam de evoluir num intervalo suficientemente grande para que a sua reutilização não provoque confusão entre um pacote novo e um pacote muito atrasado.

Se um canal ou uma rede não trocarem a ordem de entrega dos pacotes, os protocolos *stop & wait* e GBN, com emissor com janela de dimensão N e receptor com janela de dimensão 1, necessitam de tantos números diferentes quanto a dimensão da janela do emissor + 1. Assim, o *stop & wait* precisa de 2 números de sequência diferentes antes da reutilização, e o GBN necessita de $N+1$ números diferentes se a janela do receptor tiver dimensão 1. No fundo, ambos os protocolos necessitam do número máximo de pacotes possíveis em trânsito mais um. Para justificar essa necessidade de mais um, pode-se pensar no caso limite em que o emissor envia toda a janela, portanto N pacotes, e o receptor envia todos os N ACKs, mas todos eles se perdem. O emissor reemitiria a janela integralmente, mas para o receptor seriam os N pacotes seguintes pois com N números de sequência distintos, estes têm de ser reutilizados. O protocolo SR com janelas de dimensão N necessita de $2.N+1$, pois quando o receptor já recebeu todos os pacotes da janela do emissor, mas entretanto perdeu-se o ACK do mais antigo, ele ainda pode receber pacotes duplicados com número de sequência igual ao do próximo novo pacote a receber menos N , e nesse caso não saberia distinguir o novo pacote daquele que se perdeu.

Tabela 6.3: Número mínimo de números de sequência distintos requeridos pelos protocolos quando o canal ou a rede não trocam a ordem dos pacotes (N é a dimensão das janelas)

Protocolo	Número mínimo
<i>Stop & Wait</i>	2
<i>Go-Back-N</i>	$N+1$
<i>Selective Repeat</i>	$2.N+1$

No caso em que a rede possa trocar a ordem de entrega dos pacotes, o problema torna-se mais delicado, tanto mais que é também necessário assegurar que pacotes de uma conexão antiga não são confundidos com os pacotes de uma conexão mais recente entre os mesmos extremos. Por esta razão, estes protocolos usam um grande intervalo de número de sequência distintos, representado em muitos bits, para que haja uma grande folga antes de ser necessário dar a volta e reutilizar números de sequência. Por exemplo, o protocolo TCP usa números de sequência representados em 32 bits, ver o Capítulo 7.

Uma outra faceta dos protocolos de janela deslizante tem a ver com a escolha do valor a usar no temporizador que guarda a recepção dos ACKs dos pacotes emitidos, que geralmente se designa por *timeout duration* ou simplesmente *timeout*. Como se referiu várias vezes, o valor do *timeout* deve ser suficiente para acomodar o RTT com alguma folga.

Quando os protocolos de janela deslizante estão a ser usados num quadro estável, por exemplo nas extremidades de um canal ponto-a-ponto, o valor do *timeout* pode ser estimado em função das características do canal e ser constante. Quando estes protocolos estão a ser usados extremo a extremo, o valor do RTT pode ser variável e, quando a rede está próxima da saturação e as conexões são de muito longa distância,

o valor do *jitter* pode ser significativo. Nestes casos os protocolos devem estimar dinamicamente os valores do RTT e do *jitter*, e ajustar o valor do *timeout* de forma a acomodar as variações. Este assunto será de novo retomado a propósito da discussão do protocolo TCP, ver o Capítulo 7.

Para terminar a discussão das diversas facetas dos protocolos acima descritos vamos analisar o que se passa quando uma conexão é bidireccional. Nesse caso poderíamos usar duas instâncias independentes do protocolo, uma em cada sentido. No entanto, a fusão das duas instâncias de um protocolo numa implementação conjunta permite algumas optimizações. A principal dessas optimizações consiste em transportar informação de controlo nos pacotes de dados.

Assim, sempre que uma das partes envia dados para a outra, o pacote é do tipo DATA e contém um número de sequência. No entanto, esse pacote pode também enviar informação de controlo, como por exemplo um ACK do último pacote recebido no sentido contrário. Esta abordagem necessita de uma expansão do cabeçalho, introduzindo um código de operação e um número de sequência suplementares, como é ilustrado na Figura 6.21. Se não é enviada informação ou controlo na outra direcção, o código de operação NOP (ausência de informação no sentido contrário) desfaz a ambiguidade. O termo na língua inglesa para designar esta técnica é *information piggybacking* (“andar às cavalitas”) pois a informação de controlo (no sentido contrário) vai às “cavalitas” dos dados.

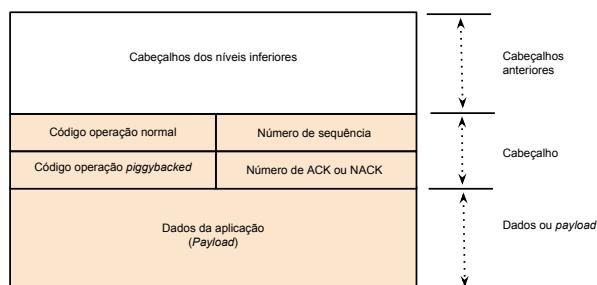


Figura 6.21: Formato das mensagens usadas pelos protocolos em conexões bidireccionais com informação *piggybacked*

6.4 Protocolos e máquinas de estado com acções

Acima descrevemos os protocolos introduzidos através de descrições informais e outras vezes algoritmos. A maioria desses algoritmos estavam estruturados como sequências de acções executadas em resposta a eventos, como a recepção de uma mensagem da rede, o disparo de um alarme, a recepção de uma mensagem da aplicação, *etc.*

Os sistemas deste tipo podem ser facilmente descritos usando máquinas de estado com acções, tratando-se basicamente de autómatos de estados. Partindo de um estado inicial, o autómato espera por um evento. Quando este ocorre, o autómato executa uma ou mais acções e transita para outro estado, onde ficará à espera do evento seguinte, e assim sucessivamente. Os eventos pertencem ao alfabeto aceite pelo autómato. Para melhor definir o seu comportamento podemos associar condições (*i.e.*, predicados) ao evento.

Uma **máquina de estados (state machine)** é um autómato com transições de estado associadas a eventos e predicados, que executa acções na transição entre estados. Um autómato deste tipo é uma forma sintética e semi-formal de descrever um protocolo e a sua evolução no tempo em função dos eventos que vão ocorrendo.

Um evento pode ser, por exemplo, receber uma mensagem ACK da rede. A condição suplementar pode ser testar se o número de sequência associado ao ACK é o que se está à espera.

O autómato é representado por um grafo em que os nós são os estados e os arcos são os eventos que ligam o estado em que o evento ocorre ao estado seguinte. Cada arco tem uma etiqueta com a indicação do evento a que está associado e as condições suplementares impostas, e também as acções a executar antes da transição de estado, separadas do evento por uma barra. Ver a Figura 6.22.

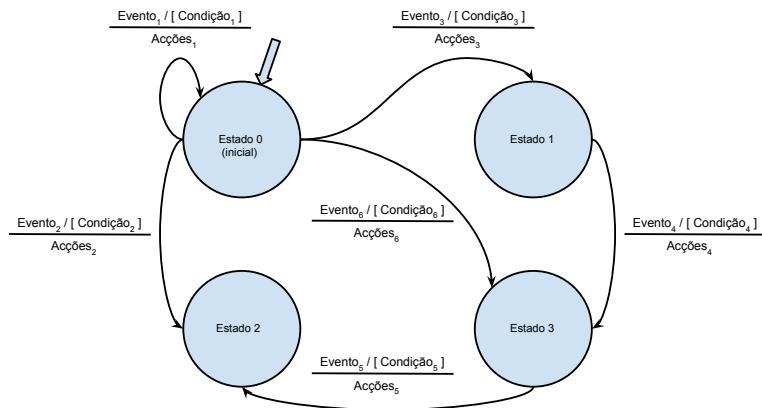


Figura 6.22: Representação gráfica de uma máquina de estados com acções

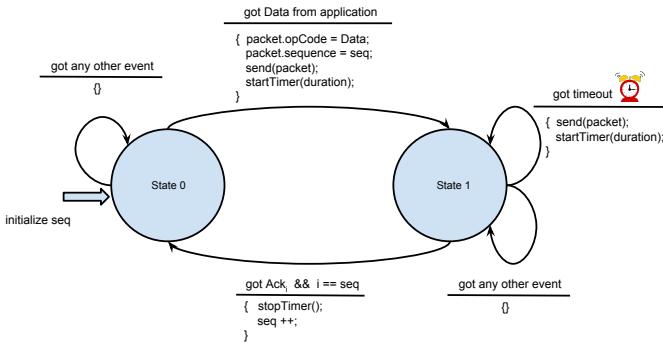
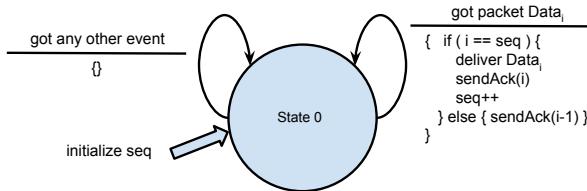
As Figuras 6.23 e 6.24 apresentam o protocolo *stop & wait* através de duas máquinas de estado, uma do emissor, e outra do receptor, respectivamente. A representação do protocolo através das duas máquinas de estados é equivalente ao algoritmo 6.1. No entanto, as máquinas de estados permitem uma visão gráfica e sintética, uma compreensão mais simples do protocolo, e a sua passagem a algoritmo com relativa facilidade.

A seguir apresenta-se um breve resumo e uma revisão dos principais conceitos introduzidos neste capítulo.

6.5 Resumo e referências

Resumo

Sempre que é necessário transferir informação de forma fiável através de uma rede ou de canais, que podem perder pacotes, é necessário utilizar mecanismos de compensação dessas falhas. Por hipótese, considera-se que os pacotes que chegarem com erros são detectados e são rejeitados. Assim, um erro num pacote é equivalente à sua perda. Com o objectivo de introduzir fiabilidade num processo de transferência de dados,

Figura 6.23: Máquina de estados do emissor *stop & wait*Figura 6.24: Máquina de estados do receptor *stop & wait*

neste capítulo analisámos uma solução, correspondente a um conjunto de protocolos, que consiste em retransmitir os pacotes que não chegaram ao receptor. Este grupo de protocolos costuma ser designado por protocolos ARQ (*Automatic Repeat ReQuest*).

Adicionalmente, estes protocolos incluem um mecanismo de controlo de fluxo (*flow control*), o qual consiste num mecanismo de extremo a extremo que adapta o ritmo de emissão de pacotes pelo emissor à capacidade de o receptor os tratar em tempo útil.

Os protocolos de transferência fiável de dados em redes de pacotes, ou em canais, que se baseiam no método de retransmissão dos pacotes perdidos, usam os seguintes mecanismos de base: pacotes de dados, pacotes de controlo, alarmes temporizados (*timeouts*) e números de sequência. Os pacotes de controlo servem para o receptor assinalar ao emissor a recepção dos pacotes enviados (ou também a sua ausência). Os números de sequência permitem que o emissor e o receptor identifiquem univocamente os pacotes.

O capítulo apresentou três protocolos que se distinguem entre si pela quantidade máxima de pacotes simultaneamente em trânsito entre o emissor e o receptor. Com o protocolo *stop & wait* o emissor envia apenas um pacote de cada vez, e só passa ao seguinte quando tem a certeza que o receptor recebeu o anterior, *i.e.*, após ter recebido uma mensagem de controlo de confirmação de recepção (ACK) com o mesmo número de sequência que a do último pacote enviado.

Os protocolos *Go-Back-N* e *Selective Repeat* mantêm vários pacotes em trânsito simultaneamente. Diferem pela forma como tratam os erros, assinalados pelos alarmes (*timeouts*) que marcam a não recepção atempada das mensagens de confirmação de recepção (ACKs). Em ambos os protocolos, os pacotes em trânsito mantêm-se num *buffer*, chamado janela do emissor, que mantém uma cópia dos mesmos, pois podem ter de ser retransmitidos.

O capítulo discute também a velocidade de transferência extrema a extremo que os diferentes protocolos permitem. Essa discussão circunscreveu-se ao caso mais simples que é quando o RTT é constante e não existem erros. O desempenho destes protocolos é muito influenciado por diversos parâmetros, nomeadamente o número máximo de pacotes em trânsito (janela de emissão) e o valor do *timeout*. A forma de fixar esses parâmetros foi também discutida.

Finalmente, foi exemplificado como os protocolos podem ser especificados de forma semi formal através de máquinas de estado, usando eventos e predicados para desencadearem as transições de estado, e a execução de acções associadas às transições de estado.

Os principais termos introduzidos neste capítulo são a seguir passados em revista. Entre parêntesis figura a tradução mais comum em língua inglesa.

Controlo de fluxo (*flow control*) Um mecanismo de controlo de fluxo é um mecanismo de extremo a extremo que adapta o ritmo de emissão de pacotes pelo emissor à capacidade do receptor os tratar em tempo útil.

Protocolos ARQ (*Automatic Repeat ReQuest protocols*) Protocolos de transferência fiável de dados, baseados na retransmissão dos pacotes cuja confirmação de recepção (ACK) pelo receptor ainda não foi recebida pelo emissor. Estes protocolos usam os seguintes mecanismos de base: pacotes de dados, pacotes de controlo, alarmes temporizados (*timeouts*) e números de sequência.

Protocolo stop & wait (*stop & wait protocol*) Protocolo em que o emissor envia um pacote e só passa a enviar o pacote seguinte quando receber a confirmação da sua recepção (ACK) vinda do receptor.

Protocolos de janela deslizante (*sliding window protocols*) Protocolos em que o emissor envia vários pacotes de avanço, mesmo sem receber os ACKs referentes aos mais antigos. Os pacotes em trânsito são mantidos num *buffer* de emissão chamado a janela do emissor. A dimensão desta janela é limitada por razões de controlo de fluxo e de controlo da saturação da rede.

Protocolo GBN (*Go-back-N protocol*) Protocolo que usa uma janela de emissão maior que 1. Quando dispara um alarme (*timeout*), o emissor recomeça a enviar por ordem todos os pacotes que ainda estão em trânsito, a começar pelo mais antigo.

Protocolo SR (*Selective Repeat protocol*) Protocolo que usa janelas de emissão e recepção maiores que 1. O protocolo trata cada pacote em trânsito de forma independente. Quando dispara um alarme (*timeout*), o emissor reenvia apenas o pacote ao qual o alarme está associado.

Máquina de estados (*state machine*) Autómato com transições de estado desencadeadas por eventos e predicados, que executa acções na transição entre estados. Um autómato deste tipo é uma forma sintética e semi-formal de descrever um protocolo e a sua evolução em função dos eventos que vão ocorrendo no tempo.

Referências

W. Stallings apresenta em [Stallings, 2013], no capítulo 7, a dedução da utilização de um canal com erros pelos protocolos *stop & wait*, GBN e SR.

Apontadores para informação na Web

- http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/index.htm – Aponta para um programa que permite visualizar o funcionamento dos protocolos *stop & wait* e GBN.

6.6 Questões para revisão e estudo

Salvo indicação em contrário, nas questões seguintes em que é necessário avaliar o desempenho dos protocolos, considere que todos os canais são ponto-a-ponto, bidimensionais, *full-duplex* (isto é, transmite nos dois sentidos simultaneamente) e considere que o tempo de processamento e o tempo de transmissão de eventuais mensagens de ACK são desprezáveis. Assuma igualmente que a velocidade de propagação do sinal nos canais é de 200.000 Km por segundo. Finalmente, despreze nos pacotes ou nas mensagens o espaço ocupado pelos cabeçalhos e considere que, por hipótese, e se não for indicado nada em contrário, não existem erros no canal.

1. A regulação do valor do *timeout* usado pelo emissor num protocolo de janela deslizante tem impacto no desempenho do protocolo em situações de erro.
 - (a) Indique os inconvenientes de um valor de *timeout* demasiado curto.
 - (b) Indique os inconvenientes de um valor de *timeout* demasiado longo.
 - (c) Indique como deve ser calculado o valor do *timeout* numa situação em que há um único canal entre o emissor e o receptor.
2. Um canal de dados tem a velocidade de transmissão de 100 Kbps e o tempo de propagação de uma extremidade à outra de 10 ms. Para mensagens de que comprimento é possível atingir uma taxa de utilização de pelo menos 50% com o protocolo *stop & wait*?
3. Considere um emissor a usar o protocolo GBN cujo código executável mantém actualizadas as seguintes variáveis:

LastPacketSent – contém o número de sequência do último pacote de dados enviado;

LastAckReceived – contém o número de sequência do ultimo ACK recebido e aceite;

MaxWindowSize – contém o tamanho máximo da janela (na verdade é uma constante).

Quais das seguintes relações são invariantes e a implementação do protocolo tem de assegurar que não são violadas? Selecione a resposta certa:

- (a) $\text{LastPacketSent} + \text{LastAckReceived} > \text{MaxWindowSize}$
- (b) $\text{LastPacketSent} + \text{LastAckReceived} \geq \text{MaxWindowSize}$
- (c) $\text{LastPacketSent} - \text{LastAckReceived} + 1 \leq \text{MaxWindowSize}$
- (d) $\text{LastPacketSent} - \text{LastAckReceived} \leq \text{MaxWindowSize}$
4. Verdadeiro ou Falso? Justifique.
 - (a) No protocolo SR é possível o emissor receber um ACK de um pacote que cai fora da sua janela de emissão.
 - (b) No protocolo GBN é possível o emissor receber um ACK de um pacote que cai fora da sua janela de emissão.
 - (c) O protocolo *stop & wait* é equivalente ao protocolo SR com o emissor e o receptor com janelas de dimensão 1.
 - (d) O protocolo *stop & wait* é equivalente ao protocolo GBN com o emissor e o receptor com janelas de dimensão 1.
 - (e) No protocolo GBN o receptor com janela igual a 1 nunca confirma a boa recepção de pacotes cujo número de sequência não seja o do pacote de que está à espera.

- (f) Com o protocolo GBN não vale a pena utilizar NACKs pois os mesmos só são úteis no protocolo SR.
 - (g) A utilização de valores de *timeout* demasiado curtos no protocolo SR impede a transferência fiável dos dados do emissor para o receptor.
 - (h) Seja qual for a situação, a utilização de uma grande janela de emissão com o protocolo GBN é sempre benéfica.
 - (i) Seja qual for a situação, a utilização de uma grande janela de emissão com o protocolo SR é sempre benéfica.
5. Um canal de dados entre dois computadores tem o débito de 1 Mbps e o tempo de propagação de 95 milissegundos. Qual a taxa de utilização do canal usando o protocolo *stop & wait* com pacotes de 10.000 bits?
 6. Um canal de dados entre dois computadores tem o tempo de propagação de 95 milissegundos. Qual o débito do canal a partir do qual a taxa de utilização do canal usando o protocolo *stop & wait* é superior a 90% com pacotes de 10.000 bits?
 7. Dois computadores A e B estão ligados diretamente por um canal com o débito de 1 Mbps e o tempo de propagação de 25 milissegundos. Escolha o tamanho mínimo do pacote (em bits) que permite uma taxa de utilização do canal de pelo menos 33,3% usando um protocolo *stop & wait*.
 8. Os computadores A e B estão a usar o protocolo GBN com uma janela de 10 pacotes de dados de 20.000 bits. O tempo de transmissão de cada pacote é de 20 milissegundos e o tempo de propagação entre A e B é de 30 milissegundos. Qual o tempo total que leva a transmitir um ficheiro com 6.000.000 bits através desse protocolo?
 9. Um canal de dados com o débito de 1,5 Mbps é utilizado para transmitir pacotes com 10.000 bits. O tempo total de propagação de uma extremidade à outra do canal é de 250 milissegundos. Os números de sequência são representados em 4 bits. Indique qual a taxa de utilização máxima deste canal com os protocolos *stop & wait* e GBN.
 10. Considere uma situação em que dois computadores estão ligados directamente por um canal ponto-a-ponto dedicado. Os computadores estão a usar o protocolo *stop & wait* para transferir dados de um para o outro. Caso o canal seja *half-duplex*, ao invés de *full-duplex*, o desempenho do protocolo é diferente?
 11. Existe uma versão do protocolo *stop & wait* que se chama “*alternating-bit protocol*”. Esta versão utiliza um único bit para codificar o número de sequência dos pacotes transmitidos do emissor para o receptor. Diga se seria possível realizar este protocolo sobre uma rede como a Internet.
 12. Considere os protocolos de janela deslizante e responda às seguintes questões de forma justificada.
 - (a) Indique pelo menos duas vantagens de num protocolo deste tipo ter o receptor com uma janela maior que 1.
 - (b) Indique se existe alguma vantagem de num protocolo deste tipo ter o receptor com uma janela maior que a do emissor.
 13. Dois computadores A e B estão ligados diretamente por um canal com o débito de 200 Kbps e o tempo de propagação de uma extremidade à outra de 75 milissegundos. A está a enviar para B pacotes com 10.000 bits de comprimento.
 - (a) Qual o número máximo de pacotes por segundo (pps) que A consegue transmitir para B transmitindo pacotes continuamente?

- (b) Qual o número máximo de pacotes por segundo (pps) que A consegue transmitir para B usando o protocolo *stop & wait*?
- (c) Qual a taxa de utilização do canal nas condições da alínea anterior?
14. Calcule o tempo total de transferência (extremo a extremo) entre dois computadores de um ficheiro com 1.000.000 bytes. Os computadores estão ligados diretamente por um canal com 10.000 Km de comprimento e portanto o tempo de propagação é de 50 ms (RTT = 100 ms). Na transferência são sempre usados pacotes com 500 bytes de dados.
- O débito do canal é de 1 Mbps e os pacotes podem ser enviados continuamente. O ficheiro considera-se transferido quando o receptor acabar de receber o último pacote. Qual a taxa de utilização do canal?
 - O débito do canal é de 1 Mbps mas depois de enviar cada pacote é necessário esperar um RTT pois está-se a usar o protocolo *stop & wait*. O ficheiro considera-se transferido quando o receptor acabar de receber o último pacote e o emissor receber o ACK dessa recepção. Qual a taxa de utilização do canal?
 - O débito do canal é de 1 Mbps mas em cada RTT só se podem enviar 10 pacotes pois está-se a usar um protocolo de janela deslizante e a janela do emissor pode ter 10 pacotes. O ficheiro considera-se transferido quando o receptor acabar de receber o último pacote e o emissor receber o ACK dessa recepção. Qual a taxa de utilização do canal?
 - O débito do canal é de 1 Mbps mas em cada RTT só se podem enviar 100 pacotes pois está-se a usar um protocolo de janela deslizante e a janela do emissor pode ter 100 pacotes. O ficheiro considera-se transferido quando o receptor acabar de receber o último pacote e o emissor receber o ACK dessa recepção. Qual a taxa de utilização do canal?
 - O débito do canal é “infinito”, isto é, o tempo de transmissão pode ser desprezado, mas em cada RTT só se podem enviar 10 pacotes pois está-se a usar um protocolo de janela deslizante e a janela do emissor pode ter 10 pacotes. O ficheiro considera-se transferido quando o receptor acabar de receber o último pacote e o emissor receber o ACK dessa recepção. Tem sentido falar em taxa de utilização de um canal com débito infinito?
 - O débito do canal é “infinito”, isto é, o tempo de transmissão pode ser desprezado, mas a janela vai aumentando dinamicamente e no 1º RTT pode-se enviar 1 pacote (isto é 2^{1-1} pacotes), durante o 2º podem ser transmitidos 2 pacotes (2^{2-1} pacotes), durante o 3º podem ser transmitidos 4 pacotes (2^{3-1} pacotes), durante o enésimo podem ser transmitidos 2^{N-1} pacotes. O ficheiro considera-se transferido quando o receptor acabar de receber o último pacote e o emissor receber o ACK dessa recepção. Tem sentido falar em taxa de utilização de um canal com débito infinito?
15. Um canal de dados tem o débito de 100 Kbps e tem 50.000 Km de comprimento. Calcule a sua taxa de utilização por um protocolo de janela deslizante com janelas com 1, 8 e 1000 mensagens, cada uma com 10.000 bits.
16. Um canal de dados tem o débito de 1 Mbps e tempo de propagação de uma extremidade à outra de 100 milissegundos.
- Suponha que se utiliza o protocolo *stop & wait* e mensagens com 10000 bits. Qual é a taxa de utilização do canal?
 - Usando o mesmo protocolo pretende-se aumentar a taxa de utilização para próximo de 100%. O que pode ser feito? A ou as soluções escolhidas têm inconvenientes? Se sim, quais são?

- (c) Suponha que se substitui o protocolo *stop & wait* por um protocolo de janela deslizante e que se continuam a usar mensagens com 10.000 bits. Para conseguir uma taxa de utilização de 90% do canal de dados, qual deveria ser a dimensão da janela do emissor?
17. Considere uma situação em que dois computadores A e B estão ligados através de uma rede. Os pacotes que transitam de A para B atravessam 2 comutadores de pacotes, C1 e C2, directamente ligados um ao outro. O canal entre C1 e C2 tem a dimensão de 1 Km e tem a capacidade de 1 Mbps. O computador A está ligado a C1 por um canal e o computador B está ligado a C2 por outro canal. Esses canais também têm a capacidade de 1 Mbps mas um tempo de propagação de 35 ms.
- Diga qual é o tempo de trânsito de A para B de pacotes com 10.000 bits de comprimento quando a rede só transmite esses pacotes.
 - Calcule a taxa de utilização da ligação entre A e B pelo protocolo *stop & wait* a utilizar pacotes com 10.000 bits de comprimento.
 - Calcule o tempo de transferência de A para B de um ficheiro com 1.000.000 bits nas condições da alínea anterior.
18. Considere uma situação em que dois computadores A e B estão ligados directamente a um comutador por canais com o débito de 1 Mbps e 5 ms de tempo de propagação. O computador A está a utilizar o protocolo *stop & wait* para enviar dados para o computador B usando pacotes com 10.000 bits. Qual a taxa de utilização pelo protocolo do canal que liga A ao comutador?
19. Considere uma situação em que dois computadores A e B estão ligados através de um canal. Os pacotes transmitidos de A para B têm dimensão constante e o tempo de transmissão T_T . O canal tem como tempo de propagação $N.T_T$.
- Qual a taxa de utilização do canal usando o protocolo *stop & wait*?
 - Usando o protocolo *stop & wait* como evoluí da taxa de utilização nos seguintes casos: $N \ll 1$, $N = 1$ e $N \gg 1$?
 - Qual a taxa de utilização do canal com o protocolo GBN usando uma janela de emissão de K pacotes?
20. Os computadores A e B estão a usar o protocolo SR com uma janela de 5 pacotes com 20.000 bits cada para transmitir dados de A para B. O tempo de transmissão de cada pacote é de 20 milissegundos e o tempo de propagação entre A e B é de 30 milissegundos. Qual o tempo total aproximado em segundos que leva a transmitir um ficheiro com 6.000.000 bits através desse protocolo?
21. Um canal de dados com a velocidade de transmissão de 1 Mbps é utilizado para transmitir pacotes com 10.000 bits. O tempo total de propagação de uma extremidade à outra do canal é de 250 ms. Indique qual a taxa de utilização deste canal com um protocolo:
- stop & wait*.
 - Janela deslizante com uma janela do emissor de 10 pacotes.
 - Janela deslizante com uma janela do emissor de 100 pacotes.
22. Dois computadores A e B estão ligados através de um conjunto de canais e comutadores. Os pacotes que transitam de A para B têm todos N bits de comprimento, atravessam C comutadores e $C + 1$ canais ponto-a-ponto *full-duplex* intermédios. Todos os canais têm o tempo de propagação T_P e o tempo de transmissão dos pacotes T_T . Considere que, por hipótese, só existe na ligação entre A e B o tráfego correspondente à transmissão do ficheiro. Diga qual o tempo de transferência de um ficheiro de dimensão F pacotes de N bits:

- (a) Usando um protocolo optimista que envia todos os pacotes em sequência.
 (b) Usando o protocolo *stop & wait*.
23. Um canal com o débito de 1 Mbps liga o computador A ao computador B. O canal tem um tempo de propagação de extremo a extremo de 20 ms. Entre os dois computadores é executado um dos protocolos de transferência de dados de A para B do tipo janela deslizante que usa pacotes de dados de 10.000 bits. Responda às seguintes questões exprimindo o resultado em percentagem.
- (a) Qual a taxa de utilização do canal entre A e B quando as janelas do emissor e do receptor são ambas iguais a 1 pacote?
 - (b) Qual a taxa de utilização do canal entre A e B quando a janela do emissor é igual a 1 pacote e a do receptor é igual a 2 pacotes?
 - (c) Qual a taxa de utilização do canal entre A e B quando a janela do emissor é igual a dois pacotes e a do receptor é igual a 1 pacote?
 - (d) Quanto tempo leva a transmitir de A para B um ficheiro com a dimensão de 10 pacotes usando a versão do protocolo indicado na alínea anterior? A transferência só termina quando o emissor receber o último ACK. O resultado deve ser expresso em milissegundos.
24. Uma instituição tem uma sede e uma sucursal cada uma com um comutador de pacotes. Na sucursal estão 10 computadores ligados directamente ao comutador da sucursal por canais de 1 Gbps e 100 metros de comprimento. Na sede existe um servidor central ligado ao comutador da sede por um canal com 10 Gbps e com também 100 metros de comprimento. O comutador da sucursal está ligado ao comutador da sede por um canal com o débito de 1 Mbps e 90 ms de tempo de propagação. Pretende-se que cada um dos computadores da sucursal envie diariamente um ficheiro de 50 Mbytes para o servidor da sede através do protocolo *stop & wait* a usar pacotes de dados com 20.000 bits. Alguém sugeriu que o melhor seria transferir um ficheiro de cada vez pois desta forma seria mais rápido, e para além disso essa opção seria suficiente para garantir a transferência diária dos 10 ficheiros. Concorda com esta solução ou há uma alternativa melhor mesmo continuando a usar o protocolo *stop & wait*?
25. Responda às seguintes questões e justifique a sua resposta.
- (a) Num protocolo de janela deslizante para que servem as mensagens de NACK enviadas pelo receptor quando os pacotes chegam fora de ordem ou estão em falta?
 - (b) Conceba um protocolo de transferência fiável de A para B baseado apenas no envio de NACKS do emissor para o receptor e indique que condições tornariam realista esse protocolo.
 - (c) As comunicações do emissor para o receptor correspondem a uma transmissão contínua que o receptor pode absorver facilmente e o canal que os liga tem uma taxa de erros desprezável. É preferível usar um protocolo baseado em ACKs ou um baseado só em NACKs para fazer as transferências?

Propostas de projecto de programação

Abaixo encontram-se a proposta de alguns projectos de programação relacionados com o protocolo *stop & wait* e o protocolo GBN.

1. Realize, por exemplo na linguagem Java, um cliente do protocolo TFTP. Este protocolo é usado para transferir ficheiros entre um cliente e o servidor através de um protocolo semelhante ao *stop & wait*. O protocolo TFTP está definido

nos RFC 1350 e RFC 2348. O cliente deverá ser capaz de transferir um ficheiro do servidor para a máquina local, onde é executado. O servidor funciona como emissor e o cliente como receptor no protocolo. O cliente será invocado através do comando:

```
java TftpGet host port filename [-s blksize]
```

<filename> é o nome do ficheiro a enviar pelo servidor

<host> é o computador do servidor

<port> porta do servidor

[-s blksize] corresponde à opção **blksize**, conforme os RFCs do TFTP

No caso de o cliente terminar a transferência com sucesso, deve afixar a seguinte informação:

- Total de bytes transferidos (dimensão do ficheiro)
 - Número total de pacotes recebidos com dados (incluindo repetições)
 - Número total de pacotes enviados com ACKs
 - Tempo total que durou a transferência
 - Número total de pacotes de dados recebidos que eram duplicados
 - Tempo médio entre o envio de um ACK e a recepção do pacote de dados seguinte sempre que foi enviado o ACK N e o bloco a seguir recebido tem o número de série N+1
2. Complete o cliente anterior de forma a que este possa também funcionar como emissor do protocolo TFTP e seja capaz de transferir um ficheiro local para o servidor. O cliente será invocado através do comando:

```
java TftpPut host port filename [-s blksize]
```

em que os parâmetros têm o significado óbvio.

3. Modifique o cliente **TftpPut** para usar o protocolo GBN. Tenha em consideração que o servidor pode ser um servidor TFTP normalizado.

Capítulo 7

O protocolo TCP

There's an old maxim that says, "Things that work persist," which is why there's still Cobol floating around.

– Autor: Vinton G. Cerf

O protocolo TCP é o protocolo de transporte actualmente mais usado na Internet. Com efeito, o mesmo é o principal suporte do protocolo HTTP e portanto sustenta a maioria das aplicações que os utilizadores da rede utilizam. Mesmo o transporte de sinal multimédia em redes TCP/IP, usado para a visualização de filmes e canais de televisão, que era até há pouco tempo predominantemente feito sobre o protocolo UDP, começou recentemente também a utilizar TCP.

O protocolo TCP foi desenvolvido na primeira metade da década de 1970, a partir de muitas experiências realizadas anteriormente, e o artigo em que o mesmo foi cientificamente divulgado data de 1974[Cerf and Kahn, 1974]. Os seus autores, Vinton G. Cerf e Robert E. Kahn, receberam em 2004 o “Turing Award”, por muitos considerado o Prémio Nobel da Informática, como reconhecimento do trabalho que desenvolveram e que envolveu também o desenvolvimento do TCP.

Como diz a citação no início do capítulo, todas as coisas que funcionam e desempenham o seu papel têm tendência a ser usadas e não são substituídas. No entanto, ao contrário da linguagem Cobol, o protocolo TCP foi sendo melhorado, de tal forma que nunca foi substituído e, apesar de haver protocolos alternativos para os mesmos objectivos, a verdade é que o TCP continua a ser o protocolo de transporte fiável mais utilizado.

Trata-se de mais um exemplo notável da virtude da separação entre a interface de uma componente e a sua implementação, um princípio tão querido da Informática. Uma outra faceta notável do protocolo consiste também na definição minimalista dessa interface, de tal forma que a mesma foi resistindo a diferentes necessidades das aplicações e a diferenças importantes na forma como essa funcionalidade é implementada, sem que a interface tenha sido modificada.

Neste capítulo são apresentadas as funcionalidades e as características de base do protocolo que têm sido constantes desde a sua introdução. Veremos também quais os mecanismos que estão na base da sua implementação. A grande maioria dos mesmos foi introduzida logo nos primórdios da sua vida. Mais tarde alguns desses mecanismos foram modificados e introduzidas extensões e alternativas. No entanto, todos esses novos mecanismos e alterações foram introduzidos sempre com a intenção de melhorar o desempenho do protocolo e são, no essencial, transparentes às aplicações que o usam,

pois não alteram a funcionalidade e filosofia essenciais do mesmo. Ou seja, são apenas melhoramentos da implementação.

A maioria desses mecanismos mais recentes, assim como outras alternativas contemporâneas, serão estudados nos capítulos que se seguem. Os mesmos estão intimamente relacionados com a própria evolução da Internet, que foi significativa desde 1974, o que teve como impacto a necessidade de adaptar o funcionamento do protocolo a essa evolução.

7.1 A interface do protocolo TCP

Como já foi referido nos capítulos 1 e 5, o protocolo TCP providencia a possibilidade de dois processos, no mesmo, ou em computadores distintos, comunicarem através de um canal lógico, designado conexão TCP. A conexão TCP permite a comunicação bidireccional e simultânea entre os dois processos. Duas versões da interface programática do protocolo já foram apresentadas nas secções 1.6 e 5.3.

O canal não disponibiliza nenhuma noção de mensagem, pois as unidades de informação transmitidas são sequências de um ou mais bytes cuja dimensão a aplicação não pode controlar. O emissor pode emitir 1000 bytes de uma só vez, mas o receptor pode ler a mesma sequência byte a byte. Não é possível ao receptor saber, através da interface do protocolo, como é que os dados que recebe foram originalmente particionados em sequências pelo emissor.

O canal é fiável no sentido em que os dados enviados pelos dois emissores, em qualquer uma das duas extremidades, chegarão ao receptor na outra extremidade pela mesma ordem e sem falhas. Caso estas duas propriedades não possam ser garantidas por qualquer motivo, a conexão não será estabelecida, ou se essa impossibilidade tiver lugar posteriormente, cada uma das partes acabará por receber uma exceção a comunicar que a conexão foi quebrada e deixou de ser possível.

O protocolo TCP incorpora dois mecanismos de controlo da velocidade de emissão: controlo de fluxo e controlo de saturação. Com efeito, ver a Figura 7.1, o protocolo TCP pressupõe que o emissor, em cada extremidade da conexão, tem um *buffer* do tipo produtor / consumidor partilhado com o nível rede do computador emissor, um *buffer* lógico constituído pelos pacotes de dados em trânsito pela rede entre os computadores emissor e receptor e, na outra extremidade, um *buffer* do tipo produtor / consumidor, partilhado entre o nível rede do computador e o programa receptor.

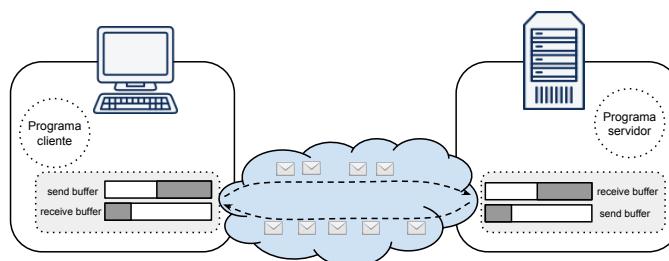


Figura 7.1: Os *buffers* envolvidos numa conexão TCP

O protocolo garante que os dados só deixam o *buffer* do emissor se (com alta probabilidade) existir espaço para os mesmos no *buffer* do receptor, usando para tal um mecanismo de controlo de fluxo que impede que um emissor rápido “afogue” um receptor lento. Como é sabido, o nível rede das redes TCP/IP não garante a entrega

de todos os pacotes pois, quer devido a erros, quer devido a saturação das filas de espera dos comutadores de pacotes, estes podem não chegar ao destino. O protocolo TCP incorpora também um mecanismo de controlo de saturação que tenta adaptar a velocidade do emissor à capacidade disponível na rede até ao destino. O objectivo é minorar a perda de pacotes devido a um débito de emissão de pacotes incomportável pela rede.

O protocolo não garante a velocidade de transferência extremo a extremo, nem sequer uma velocidade de transferência mínima. O TCP, deste ponto de vista, assume também um ponto de vista do tipo “melhor esforço”, *i.e.*, ambas as extremidades procurarão maximizar a velocidade de transferência extremo a extremo em cada sentido, mas a velocidade final será a que a rede poderá suportar em cada momento.

Veremos mais adiante que a interface do protocolo TCP permite afinar alguns parâmetros do seu funcionamento que podem influenciar o desempenho do mesmo, nomeadamente a dimensão dos *buffers* do emissor e do receptor, assim como alguns dos parâmetros que condicionam a gestão destes *buffers*.

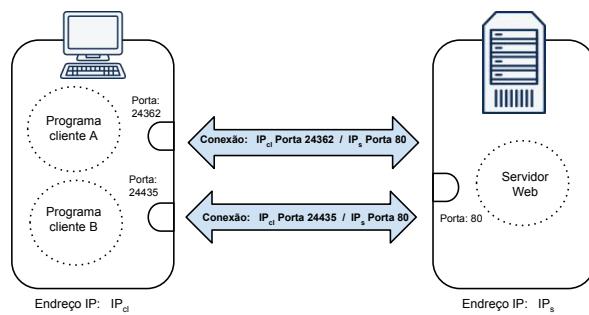


Figura 7.2: Endereços IP e portas envolvidos em duas conexões TCP entre um mesmo computador cliente e o mesmo servidor HTTP

Finalmente, importa referir que cada conexão tem de ser estabelecida antes de poder ser usada pelas duas partes, e que a mesma, como vimos no capítulo 5.3, é estabelecida entre dois sockets, caracterizados cada um por um endereço IP e uma porta. Assim, cada conexão TCP é inequivocavelmente identificada por quatro dados: o endereço IP do computador a partir do qual a conexão foi estabelecida, a porta usada na abertura, o endereço IP do computador da outra extremidade da conexão e a respectiva porta.

Por exemplo, considere-se o cenário da Figura 7.2, onde dois processos distintos no mesmo computador estabelecerem conexões TCP com um servidor Web. Do lado do servidor, ambos os sockets partilham o endereço do servidor e a porta 80, normalizada pelo protocolo HTTP. Do lado da máquina onde executa o cliente que abriu as conexões, ambos os sockets partilham o endereço IP do computador cliente, mas as portas têm de ser necessariamente distintas para que as duas conexões sejam também distintas.

O protocolo TCP disponibiliza canais lógicos fiáveis e bidireccionais entre dois processos, no mesmo ou em computadores distintos, designados conexões TCP. As conexões TCP transferem sequências de um ou mais bytes, sem qualquer relação entre a dimensão das sequências emitidas e a das recebidas, *i.e.*, o protocolo não tem a noção de mensagem.

O protocolo inclui um mecanismo de controlo de fluxo e um mecanismo de controlo de saturação da rede. Os programas utilizadores não podem solicitar ou impor os valores do débito extremo a extremo pois, deste ponto de vista, o protocolo também é baseado na noção de “melhor esforço”.

Uma conexão TCP é caracterizada pelos endereços IP e portas associadas aos sockets em ambas as extremidades. Duas conexões distintas têm necessariamente, um ou mais desses identificadores distintos.

Para perceber de forma mais completa em que consiste o protocolo teremos de penetrar na forma como está definido o comportamento das duas extremidades do canal TCP, isto é, na especificação do protocolo TCP.

7.2 Descrição do protocolo

O protocolo TCP é um protocolo de janela deslizante que pode funcionar no modo GBN (*Go-Back-N*), o modo por omissão, complementado opcionalmente com um modo semelhante ao SR (*Selective Repeat*). As unidades de informação transferidas entre as duas extremidades contêm dados e, opcionalmente, informação de controlo, e chamam-se **segmentos TCP**. O seu formato é o indicado na Figura 7.3.

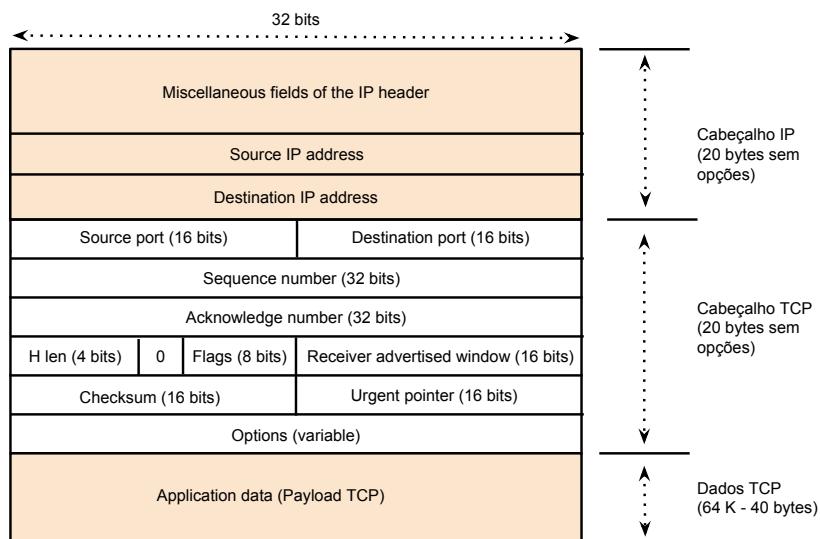


Figura 7.3: Formato de um segmento TCP

Os campos porta origem e destino, em conjunto com os campos endereço IP origem e destino do pacote IP, que encapsula o segmento TCP, permitem identificar inequivocavelmente a conexão a que o segmento pertence. Os campos número de sequência e número de ACK (do termo inglês *acknowledgement*) têm cada um 32 bits, e suportam o número de sequência dos dados quando o segmento viaja do emissor para o receptor e, opcionalmente, o número de sequência de ACK encavalitado (*piggybacked*) no sentido contrário. O mecanismo permite a cada uma das extremidades, quando envia dados, enviar também o número de sequência dos dados até aí correctamente recebidos no sentido contrário (ACK cumulativo).

O campo *flags* é constituído por um conjunto de 8 bits, em posições bem definidas, que são designados por *flags*. Cada um desses bits, se tiver o valor ‘1’, transporta informação de controlo. O posicionamento da *flag* ACK assinala que o receptor do segmento deve considerar o conteúdo do campo número de sequência ACK como válido,

e ignorá-lo no caso contrário. Assim, os segmentos TCP, dependendo dos valores do campo *flags*, podem assumir o papel de uma unidade de informação apenas com dados, apenas de controlo, ou mista, transportando dados e controlo. Várias das outras *flags* são usadas durante a abertura e fecho da conexão e ainda em circunstâncias que serão detalhadas a seguir.

O campo *checksum* contém um código de controlo de erros, calculado pelo algoritmo apresentado na secção 2.4, que protege os cabeçalhos IP e TCP e o *payload* do segmento. Naturalmente, o valor do campo *checksum* é calculado usando 0 como o seu valor anterior. Adicionalmente, alguns valores dos campos do cabeçalho IP que variam durante a viagem do pacote também têm de ter valores especiais para o cálculo do *checksum*. Para efeito deste cálculo, é habitual designar estes cabeçalhos especiais por pseudo cabeçalhos (*pseudo headers*). O formato dos pseudo cabeçalhos varia nas versões 4 e 6 do protocolo IP.

Os campos *receiver advertised window* e *urgent pointer* serão explicados posteriormente e referem-se também à informação de controlo.

O campo *header length* contém a dimensão do cabeçalho em múltiplos de 4 bytes. Assim, a maior dimensão possível do cabeçalho TCP são $15 \times 4 = 60$ bytes. Sem opções, caso de muitos segmentos, o cabeçalho tem 20 bytes, e os restantes apenas são usados quando o campo de opções existe. Várias das opções são usadas durante a inicialização da conexão e permitem a ambas as partes acordarem os diversos parâmetros que a vão posteriormente caracterizar. Um desses parâmetros, transmitido nesse momento, é o tamanho máximo dos segmentos usados na conexão. Começaremos a discussão mais detalhada do protocolo por este aspecto.

Escolha do MSS (*Maximum Segment Size*)

Apesar de teoricamente cada segmento poder ter o comprimento total de 64 Kbytes, muitos dos canais não suportam *frames* com dados dessa dimensão. Sempre que são transmitidos pacotes que não cabem no *frame* máximo possível num canal, na versão 4 do protocolo IP o pacote é decomposto em vários fragmentos, *i.e.*, o pacote é fragmentado. Assim, o segmento tem de ser enviado em diversos fragmentos, alguns dos quais poderiam não chegar ao destino, ou chegarem por uma ordem diferente da com que foram emitidos. Apesar de o protocolo IP ter mecanismos, designados por **mecanismos de fragmentação**, que realizam a fragmentação automaticamente, a mesma é hoje em dia considerada uma má prática, e a versão 6 do protocolo nem a suporta na base.

Por isso, o TCP procura encontrar o valor máximo do comprimento dos segmentos a usar durante a conexão para que não seja necessário usar fragmentação. Esse parâmetro da conexão é designado por MSS, de *Maximum Segment Size*. O MSS corresponde à dimensão máxima do *payload* e não corresponde portanto à dimensão máxima do pacote que pode atravessar um canal. Por exemplo, num canal Ethernet, que comporta *frames* com no máximo 1500 bytes de dados, o MSS é de $1500 - 40 = 1460$ bytes (20 bytes para o cabeçalho IP e 20 bytes para o cabeçalho TCP, ambos sem opções). O valor na versão 6 do protocolo IP é diferente pois a dimensão do cabeçalho IP é nesse caso maior.

O valor do MSS depende das características do conjunto dos canais atravessados pelos pacotes enviados pelas duas extremidades, que não são conhecidos pelo TCP no momento do estabelecimento da conexão. No entanto, hoje em dia, os canais que impõem maiores limitações à dimensão máxima dos *frames* concentram-se geralmente na periferia e ligam os próprios computadores à rede, ou estão perto deles. Este facto permite a cada uma das partes questionar o seu nível rede sobre o melhor valor de MSS a usar, e comunicar à outra o valor que acha mais adequado no momento da abertura da conexão. O valor final retido para o MSS da conexão será o menor dos valores comunicados, ver a Figura 7.4.

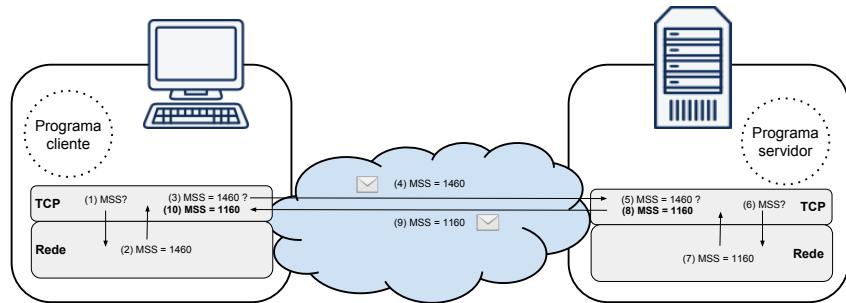


Figura 7.4: Na abertura da conexão, as partes acordam o valor do MSS (os números das legendas denotam a ordem das operações)

Como o valor do MSS escolhido na abertura da conexão não pode garantir que não existe necessidade de fragmentação, o protocolo TCP usa uma opção do protocolo IP (a única hipótese possível na versão 6 do protocolo IP) que indica que se for necessária fragmentação, a mesma deve ser rejeitada, e o emissor original do pacote avisado através de um pacote de notificação de erro. Este mecanismo, descrito no RFC 1191, para a versão 4 do protocolo IP e no RFC 1981, para a versão 6 do protocolo IP, permite afinar dinamicamente o valor do MSS. No entanto, o mesmo nem sempre funciona, pois em diversas circunstâncias os pacotes com notificações de erro são bloqueados¹. As versões mais modernas do protocolo TCP procuram afinar dinamicamente o valor do MSS quando detectam problemas, usando as técnicas descritas no RFC 4821.

Funcionamento do protocolo de janela deslizante

Por omissão, o protocolo TCP funciona segundo a estratégia GBN com valores do número de ACK cumulativos, que permitem janelas do emissor e do receptor maiores que o MSS e, opcionalmente, permitem também que o receptor guarde segmentos recebidos fora de ordem, como foi descrito na secção 6.3. No entanto, no protocolo TCP os números de sequência não identificam segmentos mas sim bytes.

Cada uma das partes fixa o seu valor inicial do número de sequência, designado ISN (de *Initial Sequence Number*), ver a Secção 7.3. O primeiro byte transmitido terá por número de sequência ISN+1, e o enésimo byte transmitido terá por número de sequência ISN+1+n. No exemplo da Figura 7.5 o número de sequência inicial é ISN+1 e o número de sequência lógico de cada byte *i* transmitido será ISN+1+*i*.

Quando é enviado um segmento, o número de sequência do segmento corresponde ao número de sequência do 1º byte contido no segmento. Por exemplo, seja 10 esse número; se o segmento contiver 5 bytes, o número de sequência do segmento seguinte será $10 + 5 + 1 = 16$ (ver a Figura 7.6). Por outro lado, o número de ACK não é o número de sequência dos bytes recebidos, mas o número de sequência do próximo byte a receber, *i.e.*, o número de sequência do último byte bem recebido + 1. No exemplo anterior, o número de ACK será 16 *i.e.*, o número de sequência do próximo byte esperado.

Cada segmento com dados pode ter de 1 a MSS bytes. O melhor rendimento é obtido com segmentos tão grandes quanto possível. No entanto, a aplicação escreve

¹O protocolo “auxiliar” do protocolo IP que permite reportar erros é o protocolo ICMP (*Internet Control Message Protocol*), já referido na Secção 3.3. O ICMP é muitas vezes bloqueado por razões de desempenho dos comutadores que detectam os erros, mas também de segurança, pois pode ser usado por um potencial atacante para obter informações sobre a configuração da rede e realizar ataques de negação de serviço aos comutadores.

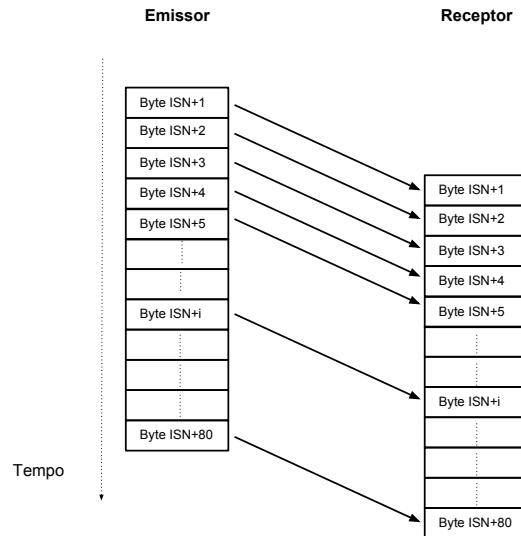


Figura 7.5: Números de sequência dos bytes no TCP

dados no canal conforme lhe é possível. Por exemplo, se a aplicação for uma aplicação do tipo sessão remota (*remote login*, *e.g.*, Telnet, *chat*), uma das mais populares aplicações usada nos primórdios da Internet, a aplicação vai obtendo bytes conforme o utilizador os escreve no teclado e o seu eco no monitor está dependente do computador remoto. Para obter o melhor desempenho possível, o TCP do lado do emissor teria de esperar até ter um número de bytes próximo do MSS. Contudo, isso poderá levar demasiado tempo, sobretudo quando a aplicação é interactiva. Como deve o TCP decidir quando deve emitir o próximo segmento?

O mecanismo usado pelo TCP é tentar esperar algum tempo para obter o máximo de bytes possível. Todavia, esse período de espera não pode exceder um valor máximo, geralmente 200 ms e, se o mesmo for ultrapassado, será enviado um segmento com os bytes disponíveis, independentemente da quantidade. A decisão de esperar ou não toma também em consideração se há ou não dados em trânsito e ainda não *acked*. O conjunto de estratégias usadas na emissão tem o nome de algoritmo de Nagle e o compasso de espera máximo foi fixado tendo em consideração vários factores, entre os quais a facilidade de implementação, que é condicionada pela granularidade dos temporizadores que é realista usar.

O socket do emissor admite que o programa, através da opção `SO_NODELAY`, force que cada escrita do programa, mesmo de um pequeno número de bytes, desencadeie o envio imediato de um segmento. A chamada `flush()` também permite ao programa solicitar o envio imediato de todos os bytes disponíveis no *buffer* de emissão. Este tipo de mecanismos são especialmente relevantes em programas interactivos. Quando os canais TCP mais não fazem que transferir dados continuamente, as opções por omissão desses mecanismos são adequadas e não necessitam de ser alteradas.

Uma questão simétrica, mas intimamente relacionada com esta, consiste em o receptor decidir quando envia um ACK. Na verdade, idealmente todos os ACKs deveriam ser enviados encavalitados nos dados enviados no sentido contrário, para evitar, tanto quanto possível, o envio de pequenos segmentos exclusivamente de controlo. Por isso, o TCP tenta esperar após a recepção de um segmento, até um máximo de 500 ms para ver se consegue transmitir o ACK encavalitado com dados enviados no sentido

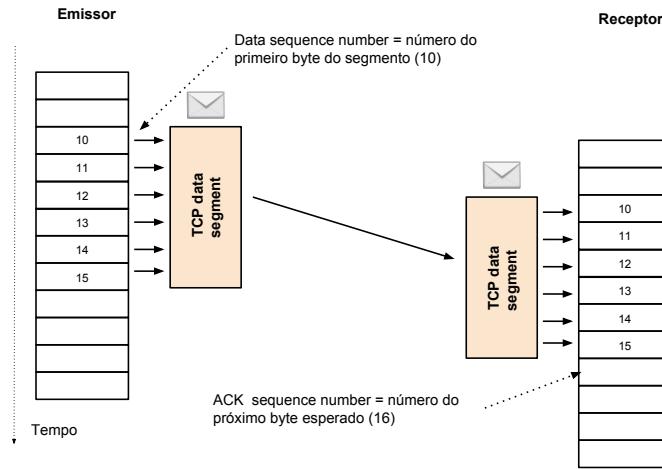


Figura 7.6: Números de sequência e de ACK dos segmentos

contrário. No entanto, se durante este compasso de espera for recebido um segundo segmento, um ACK é imediatamente transmitido mesmo que não hajam dados para transmitir no sentido contrário.

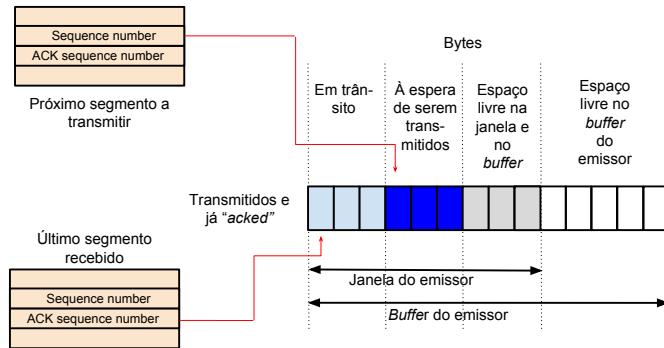
Estes mecanismos são importantes para melhorar o desempenho do protocolo, no entanto eles também pode ter um impacto negativo no desempenho de programas interactivos. As aplicações actuais (Web e nos sistemas móveis) procuram privilegiar, tanto quanto possível, o tratamento fino dos dados dos utilizadores localmente ao cliente. Esse tipo de estratégia minora este tipo de problemas, pois aumenta a dimensão das mensagens trocadas na conexão.

Como já foi dito, por omissão o protocolo funciona em modo GBN com ACKs cumulativos e o receptor pode guardar dados fora de ordem, apesar de a norma do protocolo não o impor. Em caso de alarme por ausência de ACK, são retransmitidos todos os bytes de que ainda não se receberam ACKs. Neste caso, os segmentos terão a dimensão tão próxima de MSS quanto possível, não interessando como os dados a transmitir foram escritos pelo programa emissor, ou originalmente segmentados para a transmissão. Este facto, assim como a discussão acima, mostra algumas das vantagens de o TCP não ter a noção de mensagem.

Como num protocolo desta natureza, o emissor e o receptor usam *buffers* de emissão e recepção, a dimensão dos mesmos é independente da janela de emissão para permitir o máximo de flexibilidade ao emissor e ao receptor, assim como aos respectivos programas. Um grande *buffer* de emissão permite ao TCP dar liberdade de escrita ao programa emissor e regular a dimensão dos segmentos como for mais adequado. Um grande *buffer* de recepção permite receber dados fora de ordem e acomodar alguma eventual lentidão do receptor. Ambas as dimensões têm valores por omissão que os programas podem modificar através de opções dos respectivos sockets.

É preciso, no entanto, ter em atenção que caso a rede tenha uma perda significativa de pacotes, o aumento dos *buffers*, em especial o de recepção, pode revelar-se negativo, pois pode aumentar o número de segmentos a retransmitir sempre que um alarme dispara. Este aspecto está relacionado com o mecanismo de controlo de fluxo do TCP que será discutido a seguir, e que também condiciona a dimensão da janela de emissão, *i.e.*, da quantidade de dados em trânsito.

A Figura 7.7 mostra o *buffer* do emissor e põe em evidência que a dimensão da janela de emissão não está necessariamente relacionada com a dimensão do *buffer*.

Figura 7.7: Janela de emissão e *buffer* de emissão no TCP

de emissão. No TCP, como veremos no Capítulo 8, a janela de emissão é ajustada dinamicamente, enquanto que a dimensão do *buffer* de emissão é geralmente fixa.

Naturalmente, quando o emissor envia dados tem de instalar um alarme e executar o procedimento GBN quando o alarme dispara (*timeout event*). No protocolo não existe, neste modo de funcionamento, a noção de NACK. No entanto, quando recebe dados fora de ordem, o receptor envia sem demora um ACK cumulativo. Se o emissor continuar a enviar segmentos para a frente sem executar o procedimento GBN, esse ACK será provavelmente repetido. O emissor, utiliza um mecanismo, designado FAST RETRANSMIT, que consiste em desencadear imediatamente o mecanismo GBN caso receba 3 ACKs repetidos com o mesmo número de sequência.

A Tabela 7.1, apresenta de forma mais completa, uma panorâmica da gestão do envio dos ACKs pelo receptor, evidenciando as diferentes formas de este proceder.

Tabela 7.1: Gestão do envio dos ACKs pelo receptor TCP

Estado e evento	Acção
Segmentos recebidos na ordem e já <i>acked</i> e chegou um novo segmento	Esperar até 500 ms por outro segmento ou pela oportunidade de enviar um ACK encavalitado; se o tempo de espera expirar sem ser possível enviar um ACK encavalitado, enviar um ACK sozinho
Chegou mais um segmento durante o estado anterior, ou Chegou um segmento fora de ordem, ou Chegou um segmento que fecha total ou parcialmente um “buraco”	Enviar ACK cumulativo imediatamente

O funcionamento do mecanismo FAST RETRANSMIT está ilustrado na Figura 7.8. A eficácia do mecanismo é máxima com muitos dados transmitidos para a frente, janelas de emissão grandes, janelas de recepção pequenas e transferências de grandes quantidades de dados, pois tenta acelerar a recuperação de erros e evitar retransmissões inúteis. Com pequenas quantidades de dados, ou em aplicações interactivas, o mecanismo não consegue melhorar muito a situação e com RTT elevado não consegue

evitar a retransmissão, potencialmente inútil, de segmentos recebidos fora de ordem mas potencialmente guardados pelo receptor.

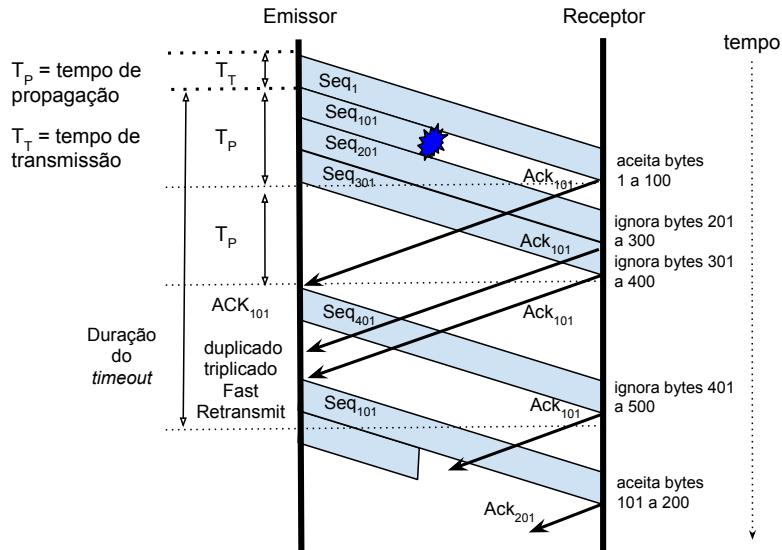


Figura 7.8: Janela de emissão e *buffer* de emissão no TCP

No protocolo TCP os números de sequência e de ACK são os números de ordem dos bytes no fluxo entre o emissor e o receptor. O número de sequência dos dados corresponde ao número de ordem do primeiro byte contido num segmento, e o número de sequência do ACK corresponde ao número de ordem do próximo byte a receber. O protocolo não impõe *a priori* nenhuma dimensão mínima, e em grande medida nem a máxima, aos segmentos que envia.

Por omissão, o protocolo é do tipo GBN com um funcionamento em que os números de ACK são cumulativos e o receptor pode guardar segmentos recebidos fora de ordem. Neste modo não existem NACKs, mas a recepção de 3 ACKs repetidos desencadeia imediatamente o procedimento GBN (FAST RETRANSMIT).

Vários mecanismos são usados para evitar enviar segmentos muito pequenos ou só com ACKs. Estes mecanismos passam pela introdução de compassos de espera limitados no tempo. Em programas interactivos os mesmos podem atrasar o progresso do protocolo, pelo que alguns podem ser desactivados. Adicionalmente, a interface também permite a manipulação da dimensão dos *buffers* de emissão e recepção.

Pelas razões atrás apresentadas, as versões mais modernas do protocolo admitem o funcionamento num modo semelhante ao *Selective Repeat*. Esta opção é comunicada à outra parte no momento da abertura da conexão. Caso ambas as partes aceitem esta forma de funcionamento, dependendo de eventuais perdas de segmentos, podem

ser enviados no campo de opções *Selective Acknowledgements*, também designados por SACKs.

Ou seja, para além de transmitir o ACK cumulativo normal, esta opção do protocolo usa o campo de opções para transmitir SACKs, *i.e.*, indicações sobre blocos de bytes correctamente recebidos para a frente, com a forma do número de sequência inicial e final de cada bloco recebido fora de ordem. Quando o receptor TCP recebe um ACK, retransmite imediatamente os dados em falta. Esta opção do protocolo está definida nos RFCs 2888 e 2889 e a sua disponibilidade é hoje em dia generalizada.

Opcionalmente, para melhorar o desempenho, o protocolo pode funcionar num modo semelhante ao *Selective Repeat*. A utilização desta opção, em conjunto com os números de sequência cumulativos normais, permite uma recuperação mais eficaz e rápida do que o mecanismo FAST RETRANSMIT, pois minora as retransmissões inúteis e repara mais rapidamente os erros.

Controlo de fluxo

O protocolo TCP inclui um mecanismo de controlo de fluxo baseado no campo *receiver advertised window*, ver a Figura 7.3. Este campo permite ao receptor TCP indicar, sempre que envia segmentos para o outro lado da conexão, qual o espaço que tem livre no seu *buffer* de recepção.

O protocolo impõe que cada emissor não deve ter dados *in-flight*, *i.e.*, transmitidos mas ainda não *acked*, em dimensão superior ao valor do campo *receiver advertised window* recebido da outra parte. Assim, o emissor tem de garantir que

$$\text{lastByteSentSequence} - \text{lastACKSequence} \leq \text{lastSeenAdvertisedWindow}$$

A Figura 7.9 mostra o cálculo deste parâmetro do lado do receptor. Caso a aplicação consuma poucos dados de cada vez, e lentamente, isso pode desencadear um fenômeno chamado *silly window syndrome*, que consiste em o emissor usar apenas segmentos de pequena dimensão, o que é mau para o desempenho do protocolo.

O algoritmo de Nagle, assim como o envio atrasado de ACKs, também foram introduzidos para combater o aparecimento deste fenômeno em situações em que o emissor transmitia, sem esperar, segmentos limitados por pequenos valores do campo *receiver advertised window*. Com efeito, sem esses mecanismos, emissor e receptor poderiam sincronizar-se e passar a utilizar apenas pequenos segmentos, mesmo com aplicações do lado do receptor que consumiam atempadamente todos os dados disponíveis no *buffer* de recepção.

Quando o valor do campo *receiver advertised window* tem o valor 0, o emissor tem de parar de enviar dados para o receptor. Se a situação persistir, poder-se-á ficar numa situação de bloqueio. Por isso o TCP do emissor usa um alarme para detectar essa situação e envia um segmento com poucos bytes quando o mesmo dispara, na esperança de que no entretanto a janela de recepção tenha alargado.

Numa situação em que o RTT é muito elevado, por exemplo de 100 ms, um emissor TCP só consegue tirar o rendimento máximo da conexão se transmitir continuamente até encher o canal lógico entre ele e o receptor. Daqui resulta que a janela de emissão tem de ter pelo menos a capacidade correspondente ao volume desse canal lógico. Isto é, deve poder transmitir continuamente até chegar o ACK do byte mais antigo transmitido, o que equivale a encher duas vezes o canal lógico entre as partes. A Tabela 7.2 ilustra o valor em bytes dessa janela, com um RTT de 100 ms, para alguns débitos de transmissão extremo a extremo significativos.

Como o campo *receiver advertised window* tem 16 bits, isso implicaria que a janela máxima de emissão do TCP fosse 64 Kbytes. Como este valor é muito baixo sempre

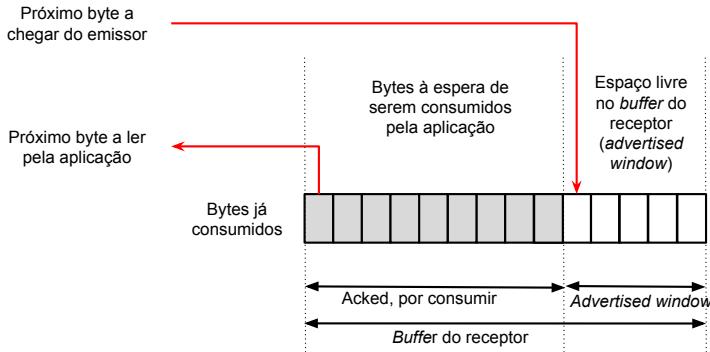


Figura 7.9: Janela de receptor e cálculo do valor de *receiver advertised window* no TCP

Tabela 7.2: Dimensão mínima aproximada da janela de emissão do TCP para aproveitar a capacidade de extremo a extremo quando o RTT é de 100 ms

Débito médio extrema a extrema	Dimensão mínima aproximada da janela TCP
100 Kbps	1250 bytes
1 Mbps	12,5 Kbytes
10 Mbps	125 Kbytes
100 Mbps	1,25 Mbytes
1000 Mbps	12,5 Mbytes

que o débito extrema a extrema seja igual ou superior a 5 Mbps (ver a Tabel 7.2), o TCP prevê uma opção chamada *window scaling option*. Durante a abertura da conexão, cada parte pode enviar à outra um valor de 0 a 14 bits no campo dessa opção que a outra parte deve usar para fazer um *shift* à esquerda dos valores do campo *receiver advertised window* que receber. Assim, se o valor desta opção para a outra extremidade for n , e o segmento recebido contiver no campo *receiver advertised window* o valor w , o receptor deve considerar que o valor do espaço livre no *buffer* de recepção da outra extremidade é $w \cdot 2^n$.

Determinação do valor do *timeout*

Como foi referido na secção 6.3, a utilização de um valor adequado do *timeout* é fundamental para o desempenho extrema a extrema dos protocolos de janela deslizante. O TCP estima o valor do RTT continuamente e usa as amostras colhidas para deduzir o valor do *timeout* a usar. Adicionalmente, o comportamento da conexão, do ponto de vista da perda de pacotes, também é usado para definir em cada momento o valor deste parâmetro.

Se a rede fosse uma rede que garantisse o tempo de trânsito de extrema a extrema, o valor do *timeout* poderia ser estimado de forma estática. Por exemplo, durante a abertura da conexão cada parte mediria o valor do RTT, e utilizaria daí para diante o dobro do valor medido como valor do *timeout* a usar.

Na verdade, o valor do RTT é variável pois o *jitter* pode ser significativo em

muitos momentos. Por outro lado, caso a conexão dure um tempo significativo, o valor médio pode ser bastante diferente se medido em períodos distintos. Tudo depende da competição pelos canais que os pacotes que transportam os segmentos da conexão encontrarem. Por exemplo, é frequente vários computadores partilharem o mesmo canal de ligação ao resto da rede. Se vários computadores abrirem simultaneamente conexões, o canal terá de ser partilhado por todas. Se o canal for limitado, em função da variação do número de conexões simultaneamente activas, a competição pelo mesmo irá também variando, assim como o estado da sua fila de espera.

Convém portanto usar um método não simplista para estimar o valor médio do RTT. Adicionalmente, para evitar retransmissões prematuras, é necessário tomar em consideração a sua variação a curto prazo.

Para estimar o valor médio do RTT, o TCP marca a hora de emissão de um segmento e mede o tempo que levou a chegar o respectivo ACK. Neste processo é necessário descartar as medidas em que o ACK foi recebido após uma retransmissão, assim como os ACKs que chegaram após ter sido aplicado algum atraso na sua emissão do lado do receptor. Por outro lado, não é possível ir simplesmente acumulando amostras e usar como RTT o seu valor médio. Por exemplo, se houvesse uma alteração sensível do RTT após as primeiras 1000 amostras, a alteração só teria repercussão na média depois de mais algumas centenas de amostras suplementares.

Para resolver este primeiro problema o TCP usa uma média ponderada entre o passado e o presente. Seja $nextRTTSample$ a última amostra obtida, e $avgRTT$ a média estimada até aí; então é possível usar o seguinte cálculo para actualizar o valor da média:

$$avgRTT = \alpha \cdot avgRTT + (1 - \alpha) \cdot nextRTTSample \quad (7.1)$$

A variável $avgRTT$ da equação acima chama-se no TCP *Smoothed RTT* ou SRTT. Este tipo de média chama-se uma **média móvel com ponderação exponencial, ou EWMA (Exponential Weighted Moving Average)**.

Para o peso do passado, o valor de α , toma-se $7/8 = 0,875$, e portanto $1 - \alpha = 0,125 = 1/8$. Estes valores foram escolhidos pois caso os valores do RTT sejam representados em inteiros, a afectação pode usar operações de *bit shift* de 3 posições, pois $8 = 2^3$. Este tipo de optimizações pode ser mais significativo em dispositivos computacionais sem muita capacidade de cálculo ou de energia. O valor de $avgRTT$ é inicializado a 3 segundos, e assim que é obtida a primeira medida, toma o valor do primeiro valor medido.

Adicionalmente, como o valor do SRTT é uma média e, por isso, muitas vezes inferior ao verdadeiro RTT, o *timeout* tem de ser calculado usando uma margem de segurança. Inicialmente o RTT medido era multiplicado por dois, mas verificou-se posteriormente que o valor escolhido era muito elevado se o RTT fosse estável e muito pequeno se o RTT variasse muito, pois quando os canais são de baixa capacidade e a sua utilização se aproxima da saturação, as filas de espera podem crescer muito e a variação do RTT ser muito maior que o próprio RTT.

Para lidar com estas situações, Van Jacobson introduziu o seguinte método para calcular o valor do *timeout* usando SRTT e a sua variação, também aliada (*Smoothed*), que é designado na fórmula abaixo SRTTVar (*Smoothed RTT Variation*):

$$timeout = SRTT + 4 \cdot SRTTVar \quad (7.2)$$

Para o cálculo de SRTTVar é usado um método semelhante ao usado para calcular SRTT:

$$SRTTVar = \beta \cdot SRTTVar + (1 - \beta) \cdot |SRTT - nextRTTSample| \quad (7.3)$$

com $\beta = 0,75$, $1 - \beta = 0,25$. O valor inicial de SRTTVar é $1/2$ do primeiro RTT medido.

Adicionalmente, sempre que se perdem pacotes, para cada uma das suas retransmissões é usado um valor de *timeout* igual ao valor anterior multiplicado por 2, até um limite superior. Assim, para evitar tanto quanto possível o envio de segmentos que provavelmente não chegarão ao destino devido a problemas na rede, o TCP retransmite os segmentos após o disparo de alarmes correspondentes a períodos de espera cada vez mais alargados.

Este tipo de gestão dos temporizadores é designada por “recurso exponencial” (*Exponential Backoff*). O RFC 6298 discute em detalhe a gestão do valor deste temporizador no TCP.

Dois aspectos suplementares merecem uma breve referência. O primeiro tem a ver com o facto de que para medir o RTT, o TCP não usa todos os segmentos e ACKs recebidos, mas apenas um subconjunto que garante que é realizada pelo menos uma medida em cada RTT. Se necessário, para realizar essa medida com precisão, é usada uma opção especial, chamada *timestamp option*, que permite colocar o valor de um relógio local ao emissor no campo de opções e receber um ACK que contém a mesma informação, transmitido assim que o segmento original chegou ao receptor. Assim, um emissor que use essa opção pode medir com rigor o RTT sem necessidade de sincronização de relógios, pois o receptor devolve imediatamente um ACK com o valor da *timestamp* enviada sem alterações, ou seja, a medição do RTT depende apenas do relógio do emissor.

O segundo aspecto tem a ver com a gestão de alarmes (*timeout*). Na secção 6.3 quando foi introduzido o protocolo SR, foi sugerido que o mesmo usa um alarme para cada pacote enviado pelo emissor. Numa conexão com um elevado RTT existem potencialmente muitos segmentos em trânsito, o que conduziria a uma grande quantidade de alarmes distintos, tanto mais que um computador pode ter várias conexões TCP activas simultaneamente. Os RFCs sugerem que cada conexão tenha apenas um alarme associado à necessidade de retransmissão. Esse alarme, tal como no protocolo GBN, está ligado ao segmento em trânsito mais antigo e ainda não *acked*, e é actualizado sempre que se recebe um ACK que faz avançar a janela “à esquerda”.

Dados urgentes

No cabeçalho dos segmentos TCP está presente o campo *urgent pointer*. O mesmo destina-se a assinalar que num segmento estão presentes dados considerados *out-of-band*, i.e., que a aplicação receptora deve tratar como dados urgentes e cujo tratamento deve ser tão rápido quanto possível. O campo *urgent pointer* é válido se a flag URG (de *URGENT*) estiver posicionada. Trata-se de um mecanismo que foi introduzido para enviar sinais urgentes à aplicação na outra extremidade da conexão, motivado pela necessidade de implementar a possibilidade de interromper a actividade normal em sessões remotas baseadas, por exemplo, no protocolo Telnet, ver o RFC 854, ou outros equivalentes (ssh, etc.).

Entre as *flags* presentes no cabeçalho, o emissor pode também posicionar uma designada PSH (de *PUSH*), que significa que os dados transmitidos no segmento devem ser transmitidos à aplicação com urgência assim que chegarem. Na interface de sockets do TCP este mecanismo não é acessível e não há nenhuma forma de um processo aceder a dados recebidos sem consumir todos os que os antecederam. Provavelmente esta opção está ligada ao facto de que a interface de sockets TCP, introduzida posteriormente ao protocolo, é orientada a *streams* e não comporta a noção de mensagens urgentes.

Ambos os mecanismos caíram em desuso e quando uma aplicação baseada em TCP necessita de usar formas de controlo deste tipo, geralmente usa duas conexões: uma para transferências normais de dados, e outra para transmissão de comandos e sinais. O protocolo FTP, ver o RFC 959, é um exemplo desta filosofia.

O protocolo TCP incorpora diversos mecanismos sofisticados para adequação dinâmica da velocidade de emissão à capacidade do receptor consumir os dados enviados (controlo de fluxo) e adaptação dinâmica do valor do *timeout* usado ao estado da rede.

Torna-se assim claro que se trata de um protocolo que tenta maximizar o desempenho através de mecanismos sofisticados de adaptação ao estado do receptor e da rede. Adicionalmente, como veremos no capítulo a seguir, incorpora ainda mecanismos de controlo da saturação da rede. Existem dezenas e dezenas de RFCs directamente ligados às diversas facetas do protocolo e à sua evolução desde que foi introduzido.

7.3 Abertura e fecho das conexões

Quando entre as partes em comunicação existe um só canal, por exemplo um canal ponto-a-ponto e *half-duplex*, é possível saber o tempo máximo durante o qual o canal pode “memorizar” pacotes (isto é, o tempo em que o sinal está em trânsito entre as extremidades do canal). Esse tempo está limitado pelo tempo de propagação. Adicionalmente, se a outra parte respondeu ao pacote p , então sabe-se que ela recebeu todos os pacotes enviados antes de p , ou nunca os receberá. Um canal não troca a ordem de entrega dos pacotes, só pode perdê-los.

No entanto, numa rede como a Internet, a rede pode memorizar pacotes durante algum tempo e trocar a sua ordem de entrega. Basta para isso que um comutador introduza atrasos extraordinários num pacote, ou que o caminho que estes seguem tenha sido alterado recentemente. Sendo assim, quando o emissor recebe a resposta ao pacote p , isso não garante que o receptor não receba posteriormente pacotes que tenham sido enviados antes de p , mas que este interpreta como tendo sido enviados depois.

Uma questão interessante que se pode colocar é: qual o tempo de vida máximo de um pacote dentro da Internet? Não é possível determinar esse valor, apenas é possível definir um valor a partir do qual a probabilidade de um pacote “perdido” ser mesmo assim entregue ser na prática nula. A esse valor chama-se MPL (*Maximum Packet Lifetime*) e no contexto do protocolo TCP, MSL (*Maximum Segment Lifetime*). Esse valor foi fixado na definição inicial do TCP no RFC 793 em 120 segundos para garantir uma segurança máxima.

Adicionalmente, por exemplo no fecho de uma conexão, é necessário a uma das partes (A), saber se a outra parte (B) está de acordo em fechar a conexão. Mas se B só puder fechar a conexão caso tenha a certeza que A recebeu a informação de que está de acordo em fechá-la, é necessário que A comunique a B que recebeu a sua resposta e tenha a certeza que B recebeu essa informação. Tal exige que B responda de novo a A e o problema não tem solução caso se possam perder pacotes.

Porque é que estas questões têm importância no protocolo TCP? Porque é necessário garantir que segmentos de uma conexão antiga não sejam confundidos com segmentos de uma conexão futura. O problema pode parecer bizarro, mas um protocolo só é robusto se resistir a todas as situações que possam suceder, por mais improváveis que pareçam. A questão prende-se intimamente com a escolha dos números de sequência iniciais, mas também com a velocidade com que os números de sequência são consumidos.

Valores iniciais dos números de sequência (ISN)

Os números iniciais de sequência, designados por ISN, de *Initial Sequence Numbers*, são estabelecidos por ambas as partes no momento do estabelecimento da conexão. Numa primeira aproximação, ambas as partes poderiam usar 0 como ISN. Na verdade isso criaria dois problemas: a entrada, por acidente, de segmentos de uma conexão antiga muito curta, numa conexão futura; e o facilitar a tentativa deliberada de um atacante de introduzir dados estranhos numa conexão. Comecemos por discutir o primeiro problema.

Existem duas formas de evitar esse problema. A primeira consiste em não reutilizar os endereços IP e as portas da conexão anterior, pelo menos até terem passado MSL segundos. Dos quatro valores que caracterizam uma conexão, apenas a porta inicial proposta pela parte que abre a conexão pode facilmente não ser reutilizada numa situação em que o mesmo computador abre sucessivas conexões para o mesmo servidor, para aceder ao mesmo serviço. Por exemplo, um cliente Web faz sucessivos pedidos HTTP (todos para a porta 80) ao mesmo servidor, usando sucessivas conexões, o que constitui uma situação frequente.

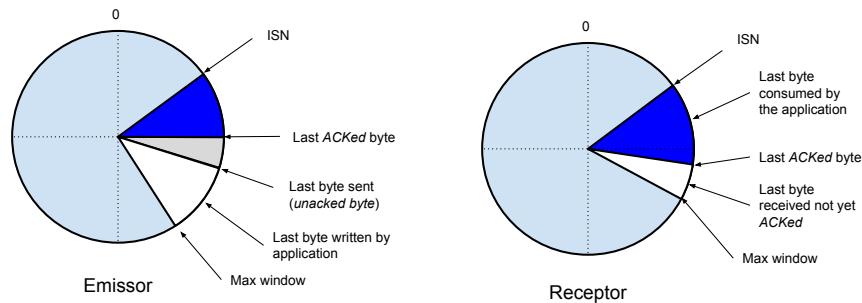


Figura 7.10: Visão da utilização dos números de sequência por cada um dos lados da conexão TCP como um relógio

A outra maneira de evitar confusão consiste em usar como ISN na conexão seguinte um valor superior ao número de sequência de qualquer dos números de sequência usados pelos bytes da conexão anterior. Este cálculo teria de ser realizado tendo em consideração que os números de sequência podem dar a volta. Na verdade os mesmos podem ser vistos como evoluindo numa circunferência, no sentido dos ponteiros do relógio, ver Figura 7.10. O ISN determina a posição inicial. Ao longo da vida da conexão os números de sequência vão crescendo no sentido dos ponteiros do relógio, até darem a volta.

Ambas as soluções (não reutilização de portas ou de números de sequência) podem ser aplicadas se as partes memorizarem alguma informação sobre as conexões antigas durante MSL segundos, e se não for possível a uma conexão consumir números de sequência a uma tal velocidade que os mesmos voltem ao início da janela em menos de MSL segundos. Infelizmente, dar a volta completa em menos de MSL segundos é “fácil” com grandes débitos, pois com um débito de extremo a extremo de 1 Gbps, os números de sequência dão uma volta completa ao “relógio” em 34 segundos, ver o RFC 1323.

Tudo indicaria ser fácil memorizar uma porta e não a reutilizar durante MSL segundos para resolver a confusão entre duas conexões distintas. Mas mesmo isso pode não ser suficiente e uma conexão suficientemente rápida pode ser induzida em erro por ela própria.

No entanto, mesmo que uma conexão leve mais do 120 segundos a dar a volta, é

sempre possível memorizar as portas ou os números de sequência das conexões fechadas nos últimos MSL segundos? A resposta é não, pois o cliente ou o servidor podem “morrer” repentinamente, perder a memória e reincarnar imediatamente a seguir. Ou outro computador poderá tomar o seu lugar e usar o mesmo endereço IP. Em ambos os casos a confusão entre duas conexões distintas pode acontecer.

Admitindo que memorizar informação sobre as conexões passadas chegaria para resolver o problema, existe uma solução fácil para o problema da “morte súbita”, que consiste em obrigar todos os computadores a levarem mais do que MSL segundos, após o seu arranque, para iniciarem a primeira conexão TCP. Dado que o MSL corresponde, por segurança, a várias dezenas de segundos, esta solução também não é em geral aceitável.

No RFC 1323 foram introduzidos um conjunto de mecanismos, baseados num relógio que se admite que não pára, mesmo quando um computador está em baixo, e na opção *timestamp* que permitem às partes trocar o valor desse relógio no início da conexão, no campo de opções. A título de parêntesis, interessa referir que o mesmo mecanismo permite também estender o número de sequência com mais 32 bits e resolver a ambiguidade com os números de sequência quando estes dão a volta, o que pode suceder facilmente em conexões de alto débito.

No entanto, nenhuma destas soluções resolvia o segundo problema, pois tornava mais fácil a um atacante adivinhar os números de sequência iniciais das conexões e enviar segmentos falsos, capazes de violar a fiabilidade das transferências.

Para tentar resolver os vários problemas enunciados, o TCP usa actualmente uma técnica de geração de números pseudo aleatórios para os valores do ISN, procura não reutilizar os números das portas usadas nas últimas conexões durante MSL segundos após o seu fecho, e usa o mecanismo das *timestamps* para troca dos valores do relógio e para proteger conexões de alta velocidade e medir mais rigorosamente o RTT.

Veremos a seguir que o TCP pode sofrer um ataque de negação de serviço durante a abertura da conexão, cuja prevenção complementa este tipo de precauções do lado da parte que aceita a conexão, usando técnicas criptográficas para calcular o ISN dessa parte.

A determinação dos ISNs (*Initial Sequence Numbers*) de uma conexão é um problema delicado quando se pretende assegurar, em todas as situações, a fiabilidade dos dados por esta transferidos. Para colmatar este problema, o TCP usa actualmente um método de determinação do ISN baseado em números pseudo aleatórios e um mecanismo de troca de valores do relógio na abertura das conexões e durante o funcionamento das mesmas.

Abertura de uma conexão

Para o estabelecimento das conexões, o TCP usa um mecanismo, chamado um “aperto de mão em 3 fases” (*three-way handshake*), baseado na troca de 3 segmentos. Para além de estes segmentos serem usados para trocar os valores dos ISNs, o MSS, etc. são também usadas as opções de passagem dos valores do relógio para evitar confusões com segmentos antigos. A utilização do *three-way handshake* é necessária para garantir que ambas as partes viram todas as informações uma da outra. Com um *two-way handshake*, a parte que aceita a conexão não teria a certeza de que duplicados de pedidos de conexão, ou duplicados de ACK, não conduziriam ao estabelecimento de mais do que uma conexão. Com efeito, dado que no protocolo TCP a parte que aceita o estabelecimento da conexão a esquece após o seu fecho, caso uma conexão durasse muito pouco tempo, um duplicado do pedido de abertura muito antigo poderia conduzir a parte que aceita a abertura da conexão a julgar que estaria perante uma conexão nova. Com o *three-way handshake*, essa parte só reconhece a conexão como aberta no fim do processo completo.

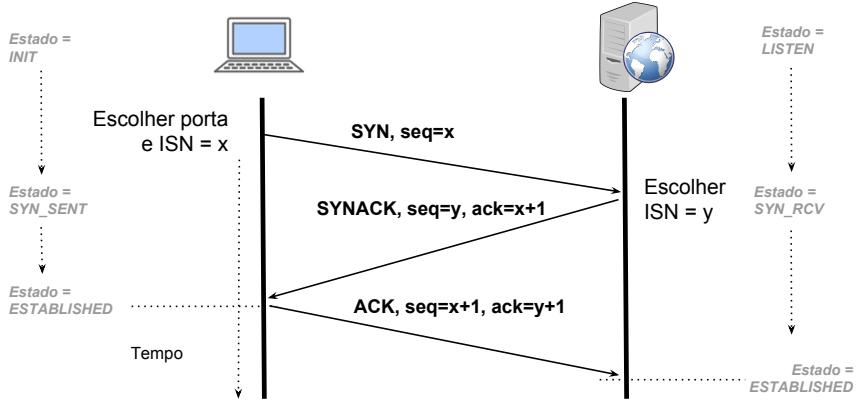


Figura 7.11: Abertura de uma conexão TCP. À esquerda a parte que solicita a abertura, à direita a que aceita a conexão

A parte que desencadeia o pedido de conexão é geralmente o cliente. A parte que atende pedidos de conexão é geralmente o servidor. Essa segunda parte criou um socket para atendimento de pedidos de conexão que está num estado, chamado LISTEN, em que aceita pedidos para novas conexões, ver a Secção 5.3. Uma conexão nessa situação costuma dizer-se que está “meio aberta”, num estado chamado “abertura passiva” (*passive open*). A parte que desencadeia a abertura da conexão diz-se que executa uma abertura activa (*active open*).

Para o estabelecimento da conexão são usadas as *flags* SYN (de *synchronize*) e ACK, como ilustrado na Figura 7.11. Inicialmente a parte que realiza a abertura activa da conexão envia um segmento com a *flag* SYN assinalada e propondo o seu número de sequência inicial (*ISN = x*). Este segmento chama-se um segmento SYN. O segmento SYN não pode transportar dados, apenas opções (MSS, *window scale*, *timestamp*, SACK suportado).

Caso a outra parte não tenha um socket no estado LISTEN associado à mesma porta, deve responder imediatamente com um segmento de erro, com a *flag* RST (RESET) posicionada. Esse segmento chama-se um segmento RST. O segmento RST também é enviado se o limite de pedidos de conexão à espera de serem atendidos pelo servidor, no mesmo socket, ultrapassou o seu limite. Este limite tem um valor por omissão que pode ser modificado através de opções do socket do servidor.

Logo que for possível atender o pedido, a parte que realizou a abertura passiva deve responder com um segmento com as *flags* SYN e ACK posicionadas, propondo o seu número de sequência inicial (*ISN = y*) e enviando um ACK do SYN recebido (com o valor $x + 1$). Este segmento chama-se um segmento SYNACK. O segmento SYNACK também não tem dados, apenas opções.

Quando recebe o segmento SYNACK, a parte que realizou a abertura activa responde com um segmento com a *flag* ACK posicionada e com o valor de ACK adequado ($y + 1$), e considera a conexão estabelecida. Este último segmento já poderia conter dados, mas a maioria das implementações enviam um segmento imediato só com o ACK. Se por acaso este último segmento ACK se perder, o próximo segmento que esta parte enviar com dados conterá necessariamente o ACK do segmento SYNACK.

Como o significado do número de sequência de um ACK é o número de ordem do próximo byte esperado, ambas as partes incrementam nos ACKs os ISNs recebidos e o número de ordem do primeiro byte transmitido por cada parte é sempre o seu ISN + 1.

O processo de abertura de uma conexão leva as partes a passarem por vários estados: CLOSED, LISTEN, SYN_RECEIVED, SYN_SENT e ESTABLISHED. O RFC do TCP contém uma máquina de estados que descreve o processo de abertura. A Figura 7.12 contém o subconjunto mais significativo dos eventos / acções e transições de estado executadas. O diagrama é, no essencial, auto-explicativo.

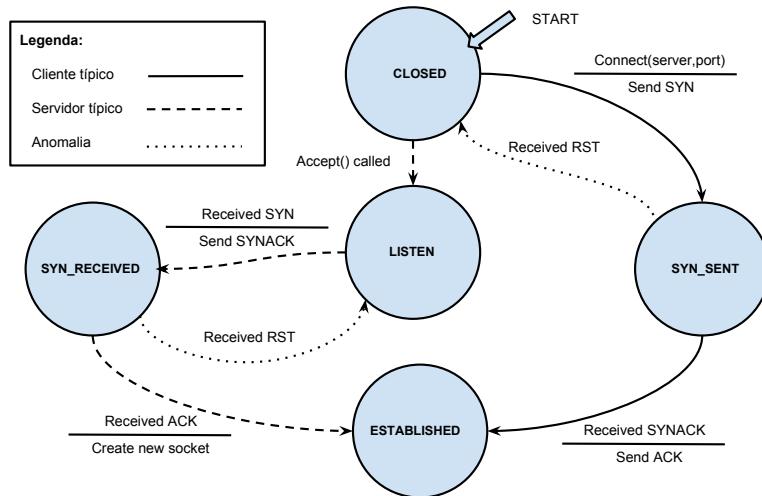


Figura 7.12: Máquina de estados simplificada correspondente à abertura de uma conexão TCP

Quando recebe o segmento SYN, a parte que está com a conexão no estado meio aberto (LISTEN), cria o estado possível para a conexão, regista as opções e afecta um *buffer* à mesma. Quando recebe o segmento SYNACK, a parte que está com a conexão no estado meio aberto (SYN_SENT), cria o estado para a conexão, regista as opções, afecta um *buffer* à conexão e considera-a aberta (ESTABLISHED).

Se o segmento SYN se perder, a parte que realiza a abertura activa reenvia-o após um *timeout*, geralmente de 3 segundos, pois no início nada se sabe sobre o RTT. Caso este primeiro segmento se perda, a abertura da conexão leva mais do que 3 segundos. Por isso, recentemente este *timeout* foi baixado para 1 segundo ou mesmo menos. Ao fim de um certo número de tentativas sem resposta, a conexão é considerada como impossível de estabelecer. Caso o segmento SYNACK se perda, a parte que realiza a abertura activa reenvia o segmento SYN, e o servidor reconhece uma conexão no estado SYN_RECEIVED e não uma nova. O temporizador para o servidor desistir e esquecer uma conexão no estado SYN_RECEIVED dura geralmente mais de 10 segundos.

Se for um *browser* Web que tenta abrir uma conexão e esta leva muito tempo a ser estabelecida, o utilizador tenta novamente aceder à página, obrigando o *browser* a abandonar a antiga conexão e a tentar abrir uma nova. A antiga tentativa é abandonada, e o cliente terá de usar necessariamente uma nova porta e outro número de sequência, pelo que o servidor verá sempre um novo pedido, e o estado da antiga será, mais tarde ou mais cedo, limpo através do temporizador que limita o tempo que uma conexão pode estar no estado SYN_RECEIVED sem se completar.

Infelizmente, se o cliente enviar uma rajada de segmentos SYN diferentes, o servidor esgotará rapidamente a sua tabela de conexões (a maioria das conexões estarão no estado SYN_RECEIVED). Trata-se de um ataque de negação de serviço, que se chama *SYN Flood Attack*, e que será discutido a seguir.

Se o último ACK se perder, como já foi referido, o cliente enviará mais tarde ou mais cedo dados que conterão esse ACK. Se não o fizer a breve trecho, o servidor esquecerá a conexão, e responderá posteriormente com um segmento RST quando receber dados referentes à mesma.

Compete à parte que abre activamente a conexão refazer os pedidos de abertura de uma conexão até conseguir uma resposta, ou desistir. A outra parte, passivamente, apenas responde a esses pedidos e espera que obtenha um ACK à sua resposta. Atendendo ao conjunto de mecanismos de discriminação de uma conexão (endereços IP, portas, ISNs e *time stamps*), não pode resultar confusão entre conexões. No máximo, existirão conexões no estado meio abertas que serão expurgadas pelo temporizador associado. Sem o *three way handshake*, o número de conexões no estado ESTABLISHED, mas inactivas, poderia ser muito maior.

Fecho de uma conexão

Quando se pretende fechar uma conexão, coloca-se o problema de saber se ainda há dados que a outra parte possa ainda querer enviar. Pretender que ambas as partes se ponham de acordo sobre um fecho simultâneo da conexão, porque já não têm nada a dizer uma à outra, cria um problema delicado de consenso que, como já referimos, é difícil de resolver quando a rede pode perder pacotes. Assim, é comum usarem-se protocolos de terminação abrupta de conexões, com eventual perda de dados já escritos por uma das partes, mas ainda não recebidos pela outra. Este efeito é possível no TCP usando segmentos RST que terminam abruptamente conexões.

No entanto, nos casos normais, para evitar terminações abruptas de conexões, o TCP usa um protocolo em que as duas partes de uma conexão fecham de forma independente cada metade da mesma. Assim, se uma das partes já não tem nada a transmitir, e tem a certeza que a outra parte já recebeu os seus dados, fecha a conexão do seu lado, mas pode continuar a ler os dados que a outra parte lhe continuar a enviar. Quando esta tiver a certeza que todos os seus dados foram bem recebidos, fecha por sua vez a conexão. Os sockets TCP dispõem da chamada `shutdown()`, que permite fechar um socket só para escrita, só para leitura, ou para ambas as funcionalidades. A chamada `close()` impede qualquer futuro acesso ao socket pelo processo que a invocar, e implicitamente invoca a funcionalidade de `shutdown()` se este for o único processo com acesso ao socket.

Para fechar uma conexão, cada uma das partes envia um segmento com a *flag* FIN (de *Finalize*) assinalada. Este segmento tem de ser *acked* como normalmente. O mesmo pode ser reenviado um certo números de vezes se necessário. A outra parte, quando desejar, deve fechar a conexão executando o protocolo simétrico. Ver a Figura 7.13.

Associado ao fecho de uma conexão existe igualmente uma máquina de estados que está ilustrada de forma simplificada na Figura 7.14. Dado que ambas as partes podem decidir fechar a conexão de forma independente, em momentos diferentes, existem três possibilidades diferentes de fechar a conexão:

Um lado fecha primeiro ESTABLISHED → FIN_WAIT1 →
FIN_WAIT2 → TIME_WAIT → CLOSED

O outro lado fecha primeiro ESTABLISHED → CLOSE_WAIT →
LAST_ACK → CLOSED

Ambos fecham simultaneamente ESTABLISHED → FIN_WAIT1 →
CLOSING → TIME_WAIT → CLOSED

Como se pode verificar pelo diagrama de estados (Figura 7.14) e pelas transições acima, a parte que desencadeia o fecho da conexão (geralmente o cliente) passa pelo estado TIME_WAIT, onde permanece $2 \times MSL$ segundos, um período de tempo significativo (com dezenas de segundos). Este mecanismo garante que a conexão que

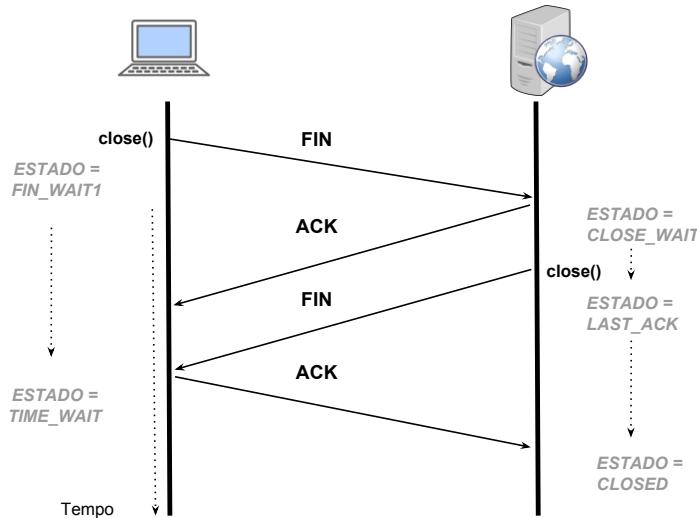


Figura 7.13: Protocolo de fecho de uma conexão TCP

foi fechada é memorizada durante $2 \times \text{MSL}$ segundos suplementares, o que assegura que não pode haver confusão entre conexões devido a segmentos atrasados.

Todas estas precauções com o fecho das conexões levam a um fecho relativamente “longo” e que guarda estado sobre uma conexão, para todos os efeitos já inactiva, durante muito tempo. Isso pode tornar-se problemático num servidor Web que aceita vários milhares de conexões curtas por minuto. Por isso, alguns desses servidores fecham as conexões dos clientes usando segmentos com a flag RST posicionada para fechar abruptamente as conexões. Isso pode não implicar necessariamente a perda de dados dado que o protocolo HTTP, ao nível aplicacional, permite controlar se há ou não mais dados para receber.

O processo de abertura e fecho de uma conexão é relativamente pesado, envolvendo a troca de pelo menos 3 segmentos para a abertura e 4 para o fecho. A abertura é também impossível num período inferior a um RTT, durante o qual não é possível as partes trocarem dados. Se uma conexão for usada para transferência de pequenas quantidades de dados, o desperdício pode ser elevado.

Por isso, protocolos cliente / servidor como o do DNS realizam as suas transações sobre UDP, pois a resposta funciona como ACK do pedido. Atendendo a que os pacotes se podem perder, este tipo de solução pode resultar na repetição das operações, o que não é grave se estas não alterarem o estado do servidor (*e.g.*, leituras ou outras operações idempotentes).

O protocolo TCP é fiável, evita esses problemas e tem grandes vantagens quando ambas as partes pretendem transferir quantias significativas de dados. É claro que não se pode deixar de referir, que se uma conexão terminar abruptamente, o emissor não tem a certeza sobre se os dados chegaram ou não ao receptor.

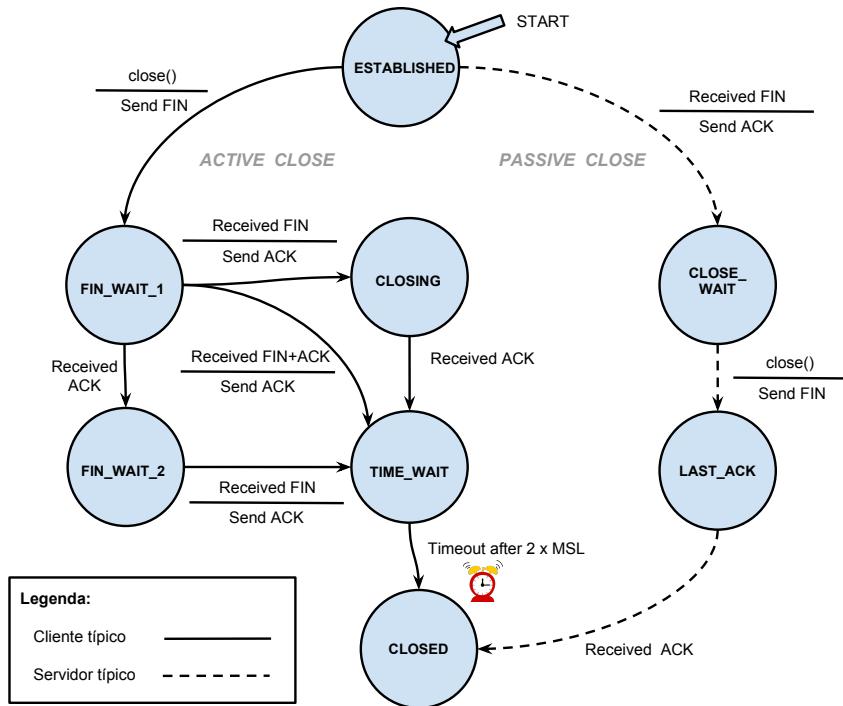


Figura 7.14: Máquina de estados correspondente ao fecho de uma conexão TCP

Ataque SYN Flood

Na implementação inicial do protocolo TCP, assim que a parte com conexões em estado passivo recebia o segmento SYN, criava todo o estado local necessário à futura conexão TCP. Isso incluía não só criar uma entrada na tabela de conexões, como afectar-lhe os dois *buffers*, de recepção e emissão.

Como na Internet é possível enviar pacotes com endereço e porta origem falsos sem que isso impeça muitas vezes o pacote de chegar ao destino, a partir de certa altura foi descoberta uma forma de atacar servidores TCP bem conhecidos, enviando-lhes uma grande quantidade de segmentos SYN com origem (do ponto de vista do servidor) em muitos clientes diferentes pois os diferentes endereços origem eram falsos.

O resultado era que o servidor esgotava rapidamente as tabelas de conexões, todas no estado SYN_RCV. Naturalmente, o atacante nem sequer recebia os segmentos SYNACK, mas mesmo que recebesse não estaria interessado em responder.

Trata-se de um ataque de negação de serviço, designado ataque por inundação de SYNs (*SYN Flood Attack*), que impede rápida e facilmente o servidor de servir clientes normais, sem custos significativos para o atacante.

Mais uma vez se revelou o inconveniente de ser possível usar endereços origem falsos para realizar ataques de negação de serviço, porque os ISPs não são obrigados a controlá-los. Com efeito, um ISP que realize esse controlo, não se está a defender, apenas está a contribuir para a defesa dos clientes dos seus concorrentes. Este tipo de situações costuma chamar-se de “tragédia dos comuns”.

Para confrinar este ataque, foi desenvolvida uma técnica, designada *SYN Cookies*, descrita no apêndice do RFC 4987, que consiste em o servidor não memorizar senão a informação mínima sobre o cliente e devolver-lhe o seu ISN calculado criptograficamente.

mente, através da seguinte função:

$$\text{server_ISN} = F(\text{connection parameters}, \text{Secret_Key})$$

em que F é uma função de *hash* criptográfica e o parâmetro *Secret_Key* é um segredo que só o servidor conhece. O ISN enviado é assim computacionalmente impossível de reproduzir por alguém que desconheça a chave secreta do servidor.

Quando o cliente responder ao segmento SYNACK com um segmento ACK, se responder, o servidor decrementa o valor do campo ACK recebido, procura na sua tabela especial o pedido de conexão enviado com o SYN, e completa então o processo de estabelecimento da conexão pois sabe que o cliente existe.

Muitos ataques de negação de serviço são combatidos decrementando o ritmo por unidade de tempo dos novos eventos suspeitos que são aceites. Os que ultrapassarem esse ritmo são ignorados. O problema fundamental deste tipo de estratégia é que é necessário minorar os pedidos de clientes honestos que são suprimidos, por serem confundidos com pedidos de atacantes (chamados falsos positivos). A estratégia descrita acima permite identificar, com elevada probabilidade de não errar, os pedidos do atacante.

O RFC 4987 discute este ataque e o conjunto de técnicas que podem ser usadas para o combater.

Análise simplificada do desempenho do TCP

O desempenho do TCP é muito influenciado pelos mecanismos de recuperação de erros e de controlo da saturação da rede, que serão discutidos no capítulo 8. No entanto, a menos da influência desses mecanismos, é desde já possível estimar qual o desempenho máximo teórico.

Com efeito, admitindo que é possível desprezar o tempo de transmissão dos segmentos e que o RTT é significativo, *i.e.*, num cenário que os canais têm débitos elevados (*e.g.*, superiores a dezenas de Mbps) e o RTT é algumas dezenas de milissegundos, e admitindo também que não há perda de segmentos nem de ACKs, a velocidade média de transferência, ou débito médio, extremo a extremo, é dado por:

$$\bar{V} = \frac{\bar{W}}{\bar{RTT}}$$

ou seja, o débito médio é a dimensão média em bits da janela a dividir pelo RTT médio. Naturalmente, este débito também não pode ser superior ao débito do canal que liga o emissor à rede.

Assim, numa situação ideal, sem erros, sem perda de segmentos e sem interferência dos mecanismos de controlo de fluxo, a janela do emissor, assim como o RTT, têm uma influência decisiva sobre o desempenho do protocolo. Nestes cenários são necessárias janelas de emissão de dimensão elevada se os débitos e o RTT forem significativos, é necessário que o receptor as permita utilizar (o mecanismo de controlo de fluxo limita a janela do emissor ao espaço livre na do receptor) e é também necessário que a implementação de todos os mecanismos do TCP consiga funcionar atempada e sustentadamente.

7.4 Resumo e referências

Resumo

O protocolo TCP disponibiliza canais lógicos fiáveis e bidireccionais, entre dois processos, no mesmo ou em computadores distintos, designados conexões TCP. As conexões

TCP transferem sequências de um ou mais bytes, sem qualquer relação entre a dimensão das sequências emitidas e a das recebidas, *i.e.*, o protocolo não tem a noção de mensagem.

O protocolo inclui um mecanismo de controlo de fluxo e um mecanismo de controlo de saturação da rede. Os programas utilizadores não podem solicitar ou impor os valores do débito extremo a extremo pois, deste ponto de vista, o protocolo também é baseado na noção de “melhor esforço”.

Uma conexão TCP é caracterizada pelos endereços IP e portas associadas aos sockets em ambas as extremidades. Duas conexões distintas têm necessariamente um ou mais desses identificadores distintos.

No protocolo TCP os números de sequência e de ACK são os números de ordem dos bytes no fluxo entre o emissor e o receptor. O número de sequência dos dados corresponde ao número de ordem do primeiro byte contido num segmento, e o número de sequência do ACK corresponde ao número de ordem do próximo byte a receber. O protocolo não impõe *a priori* nenhuma dimensão mínima, e em grande medida nem a máxima, aos segmentos que envia.

Por omissão, o protocolo é do tipo *Go Back N* com um funcionamento em que os números de ACK são cumulativos, e o receptor pode guardar segmentos recebidos fora de ordem. Neste modo não existem NACKs, mas a recepção de 3 ACKs repetidos desencadeia imediatamente o procedimento GBN (FAST RETRANSMIT).

Vários mecanismos são usados para evitar enviar segmentos muito pequenos ou só com ACKs. Estes mecanismos passam pela introdução de compassos de espera limitados no tempo. Em programas interactivos os mesmos podem atrasar o progresso do protocolo pelo que alguns podem ser desactivados. Adicionalmente, a interface também permite a manipulação da dimensão dos *buffers* de emissão e recepção.

Opcionalmente, para melhorar o desempenho, o protocolo pode funcionar num modo semelhante ao *Selective Repeat*. A utilização desta opção em conjunto com os números de sequência cumulativos normais permite uma recuperação mais eficaz e rápida do que o mecanismo FAST RETRANSMIT, pois minora as retransmissões inúteis e repara mais rapidamente os erros.

O protocolo TCP incorpora diversos mecanismos sofisticados para adequação dinâmica da velocidade de emissão à capacidade do receptor consumir os dados enviados (controlo de fluxo) e adaptação dinâmica do valor do *timeout* usado ao estado da rede.

Torna-se assim claro que se trata de um protocolo que tenta maximizar o desempenho através de mecanismos sofisticados de adaptação ao estado do receptor e da rede. Adicionalmente, como veremos no capítulo a seguir, incorpora ainda mecanismos de controlo da saturação da rede. Existem dezenas e dezenas de RFCs directamente ligados às diversas facetas do protocolo e à sua evolução desde que foi introduzido.

A determinação dos ISNs (*Initial Sequence Numbers*) de uma conexão é um problema delicado, quando se pretende assegurar, em todas as situações, a fiabilidade dos dados por esta transferidos. Para colmatar este problema, o TCP usa actualmente um método de determinação do ISN baseado em números pseudo aleatórios e um mecanismo de troca de valores do relógio na abertura das conexões e durante o funcionamento das mesmas. Adicionalmente, no fecho de uma conexão procura-se assegurar, durante um lapso de tempo significativo, que a mesma não é esquecida, e as suas portas reutilizadas.

O processo de abertura e fecho de uma conexão é relativamente pesado, envolvendo a troca de pelo menos 3 segmentos para a abertura, e 4 para o fecho. A abertura dura pelo menos um RTT, durante o qual não é possível as partes trocarem dados. Se uma conexão for usada para transferência de pequenas quantidades de dados, o desperdício pode ser elevado.

Por isso, protocolos cliente / servidor como o do DNS realizam as suas transacções sobre UDP, pois a resposta funciona como ACK do pedido. Atendendo a que os pacotes se podem perder, este tipo de solução pode resultar na repetição das operações, o

que não é grave se estas não alterarem o estado do servidor (*e.g.*, leituras ou outras operações idempotentes).

O protocolo TCP é fiável, evita esses problemas e tem grandes vantagens quando ambas as partes pretendem transferir quantias significativas de dados. É claro que não se pode deixar de referir que se uma conexão terminar abruptamente, o emissor não sabe se os dados chegaram ou não ao receptor.

Os segmentos TCP, ver a Figura 7.3, podem transportar simultaneamente dados e informação de controlo. A codificação da informação de controlo usa vários campos, o conjunto de *flags* que figura na Tabela 7.3 e o conjunto de opções que figura na Tabela 7.4.

Tabela 7.3: Tabela de *flags* do cabeçalho TCP

<i>flag</i>	Valor	Descrição
URG	0x20	Urgent data present
ACK	0x10	ACK field is valid
PUSH	0x08	Push data to the receiver
RST	0x04	Reset connection
SYN	0x02	Synchronise connection
FIN	0x01	Finalize connection
CWR	0x80	Reduce cong. window (congestion related, RFC 3168)
ECE	0x40	ECN Echo (congestion related, RFC 3168)

Tabela 7.4: Tabela de opções do cabeçalho TCP

Opção	Descrição
1	Maximum segment size
2	Window scale
3	Selective ACK permitted
4	Timestamp

Alguns dos termos introduzidos com um significado específico no contexto do TCP são a seguir passados em revista.

Receiver Advertised Window Campo do cabeçalho TCP que permite ao emissor de um segmento indicar à outra parte qual o espaço livre no seu *buffer* de recepção, para que desta forma limite a quantidade máxima de bytes em trânsito que lhe são dirigidos.

FAST RETRANSMIT Depois de emitir vários segmentos, o emissor quando recebe três ACKs seguidos com o mesmo número de sequência, conclui que o receptor perdeu algum segmento e inicia o processo de retransmissão GBN.

ISN *Initial Sequence Number*. Número de sequência do primeiro segmento enviado por cada extremo da conexão TCP. O seu valor é calculado de forma aleatória de maneira a minimizar a probabilidade de um segmento atrasado, de uma antiga conexão, ser confundido com um segmento da conexão que se inicia. Adicionalmente, esta forma de escolha do valor do ISN tenta também minimizar a probabilidade de um atacante acertar em quais são os números de sequência usados por uma conexão.

MSL *Maximum Segment Lifetime*. Tempo de vida máximo de um segmento dentro da rede. Este conceito tem relevância para segmentos anormais cuja entrega é atrasada pela rede. É usado para assegurar que um segmento atrasado não pode ser confundido com um segmento emitido posteriormente, mas com o mesmo número de sequência.

MSS *Maximum Segment Size*. Dimensão máxima dos dados (*payload*) de um segmento de uma conexão TCP. O seu valor é fixado de tal forma que se procura que um segmento caiba sempre dentro de um *frame* em todos os canais que tem de atravessar.

SACK *Selective ACK*. Opção do cabeçalho TCP que permite ao receptor sinalizar a correcta recepção de um ou mais segmentos fora de ordem.

Syn-flood Attack Forma de ataque em que o atacante inunda a vítima com segmentos SYN com endereço de origem falso e não responde aos SYNACKs, obrigando-a a esgotar o limite de conexões meio abertas.

Three Way Handshake Protocolo envolvendo o envio de 3 segmentos usado pelo TCP para abrir uma conexão.

Urgent Pointer Campo do cabeçalho TCP que permite ao emissor assinalar ao receptor a presença de dados *Out-of-Band* no segmento. Trata-se de um mecanismo usado predominantemente por aplicações interactivas como Telnet ou ssh.

Referências e apontadores

Vinton Gray Cerf e Robert E. Kahn receberam em 2004 o Prémio Turing Award da ACM (*Association for Computer Machinery*) e passamos a citar: “for pioneering work on internetworking, including the design and implementation of the Internet’s basic communications protocols, TCP/IP, and for inspired leadership in networking.” Ambos são os autores de um artigo, [Cerf and Kahn, 1974], que introduziu formalmente na literatura científica o protocolo TCP.

As publicações sobre o protocolo TCP e a sua evolução são inúmeras. No entanto, o livro [Stevens and Wright, 1994] continua a ser considerado a obra mais completa sobre os mecanismos do protocolo TCP, pelo menos até à publicação da sua última edição, pois o seu primeiro autor faleceu em 1999. O livro não só apresenta os conceitos fundamentais como os ilustra através de inúmeras experiências.

Existem imensas ferramentas que permitem medir o desempenho do protocolo TCP, nomeadamente os programas **iperf**, já referido, e **TTCP** (*Test TCP*). Richard Stevens desenvolveu um programa especial, o programa **sock**, para teste e experimentação com as diferentes opções dos protocolos UDP e TCP acessíveis via os respectivos sockets. O *packet sniffer wireshark* tem imensas opções para análise dos segmentos TCP e estudo do comportamento de uma conexão TCP. O programa **netstat**, disponível em praticamente todos os sistemas de operação, permite obter informações sobre as conexões TCP do computador onde o mesmo executa. **Web10G** é um módulo para o núcleo dos sistemas Linux que permite aceder directamente ao valor de todas as variáveis que caracterizam as conexões TCP activas. É especialmente adequado para o estudo dos factores que podem estar a limitar o desempenho das conexões TCP de um sistema Linux.

Apontadores para informação na Web

- <http://ietf.org/rfc.html> – É o repositório oficial dos RFCs, nomeadamente dos citados neste capítulo.
- <http://en.wikipedia.org/wiki/Ttcp> – Contém informação sobre o programa TTCP.

- <http://en.wikipedia.org/wiki/Iperf> – Contém informação sobre o programa iperf.
- <http://www.icir.org/christian/sock.html> – Contém informação sobre o programa sock de Richard Stevens.
- <https://www.wireshark.org> – Contém informação sobre o wireshark.
- <http://en.wikipedia.org/wiki/Netstat> – Contém informação sobre o programa netstat para diferentes sistemas de operação.
- <http://www.web10g.org> – Contém informação sobre o módulo Web10G.

7.5 Questões para revisão e estudo

1. Nos sistemas de operação, os processos em execução têm um número, geralmente chamado o PID (*Process Identifier*) do processo. Alguém sugeriu que esse número era mais interessante para discriminar as conexões TCP do que as portas. Acha a sugestão boa? Justifique a sua resposta.
2. Verdade ou mentira? Justifique a resposta.
 - (a) Para abrir e fechar as conexões, o protocolo TCP usa segmentos de controlo com cabeçalhos de formato diferente dos cabeçalhos dos segmentos de dados.
 - (b) Todas as conexões TCP usam 0 como número de sequência inicial.
 - (c) No protocolo TCP o número de sequência de um segmento é igual ao número inicial (ISN) mais o número de segmentos já emitidos.
 - (d) O protocolo TCP garante que a dimensão da janela do emissor é constante.
 - (e) O protocolo TCP exige que a dimensão da janela do receptor seja sempre constante.
 - (f) Quando executa o procedimento GBN, o protocolo TCP reemite exactamente os mesmos segmentos, com os mesmos números de sequência, que os que já tinha emitido previamente.
3. Verdade ou mentira? Justifique a resposta.
 - (a) Os canais TCP não garantem a entrega ao receptor dos pacotes transmitidos pelo emissor, nem sequer na ordem pela qual os mesmos foram transmitidos.
 - (b) Os canais TCP implementam um canal lógico de transmissão entre o emissor e o receptor cujo débito é constante.
 - (c) Uma conexão TCP é fiável de extremo a extremo, isto é, não se perdem dados, porque os comutadores de pacotes da rede garantem que todos os pacotes que lhes chegam são entregues intactos.
 - (d) Uma conexão TCP é assegurada pela rede através de uma concatenação de sub-canais físicos, chamada um circuito, que liga de forma dedicada o emissor ao receptor, e desta forma garante que todos os segmentos TCP chegam sempre ao destino.
 - (e) Uma conexão TCP é assegurada pelo sistema de operação dos computadores em diálogo, através de um protocolo que pressupõe que a rede pode perder e trocar a ordem dos pacotes IP.
4. Verdade ou mentira? Justifique a resposta.

- (a) Um computador A está a enviar um ficheiro para o computador B através de uma conexão TCP. Supondo que o computador B também tem dados para enviar para A, então B envia necessariamente para A dois tipos de segmentos: segmentos com dados e segmentos com ACKs.
- (b) O valor do campo *Receiver Advertised Window* do cabeçalho TCP nunca se altera nos diferentes segmentos usados durante toda a duração de uma conexão TCP.
- (c) Um computador A está a enviar um ficheiro para o computador B através de uma conexão TCP. Supondo que o número de sequência de um segmento enviado de A para B é N, então o número de sequência do próximo segmento enviado por A é necessariamente N+1.
- (d) Um computador A está a enviar um ficheiro muito grande para o computador B através de uma conexão TCP. Supondo que o computador B não tem dados para enviar para A, então B não envia ACKs para A pois não pode enviar ACKs encavalitados em segmentos com dados.
5. Verdade ou mentira? Justifique a resposta.
- (a) Numa conexão TCP, o *timeout* do emissor é apenas activado no fim da transmissão de toda a janela.
- (b) Num pacote IP são encapsulados um ou mais segmentos TCP.
- (c) Numa conexão TCP, o valor do *timeout* do emissor é quase sempre constante.
- (d) Numa conexão TCP, quando não se está a usar a opção SACK do protocolo, a janela do receptor tem sempre a dimensão do MSS.
- (e) Numa conexão TCP, o mecanismo FAST RETRANSMIT foi introduzido porque se pressupõe que a janela do receptor tem sempre a dimensão do MSS.
6. Indique quais das seguintes razões justificam que o número de sequência inicial de cada um dos lados de uma conexão TCP seja um número pseudo-aleatório. Justifique a sua resposta.
- (a) Por segurança, na medida em que assim os atacantes não conseguem adivinhar esse número de sequência.
- (b) Para evitar que algum pacote antigo possa ser confundido como um segmento da nova conexão.
- (c) Para identificar o momento em que a conexão foi aberta.
- (d) Porque a lógica dos protocolos de janela deslizante assim o exige.
- (e) Para diminuir a eficácia de um ataque de negação de serviço do tipo *SYN Flood Attack*.
7. O emissor TCP emite segmentos logo que recebe dados escritos pela aplicação? Se não é o caso, indique porque razão não é assim, e como é que este procede.
8. Diga para que serve a opção SO_NODELAY aplicada a um socket TCP.
9. O receptor TCP emite segmentos só com ACKs logo que recebe segmentos vindos da rede? Se não é o caso, indique porque razão não é assim, e como é que este procede.
10. Diga em que consiste a anomalia *silly window* do TCP.
11. O protocolo TCP não suporta mensagens de controlo NACK. No entanto, se considerarmos todos os mecanismos obrigatórios e opcionais, o TCP dispõe de dois mecanismos que substituem os NACKs. Quais são e como funcionam?

12. Um computador A abriu uma conexão para o computador B e fez o *download* de um ficheiro com 1024 bytes. Quantos segmentos trocaram A e B para realizar esta operação sabendo que o MSS da conexão era de 1460 bytes e que não se perderam segmentos?
13. Na situação da questão anterior, sabendo que o tempo de propagação medido de A para B era de 100 ms, e era idêntico ao que B mediou para A, indique quanto tempo levou a transferência (desde ao início da abertura da conexão até à recepção do último byte do ficheiro).
14. A regulação do valor dos alarmes (*timeouts*) usados pelo emissor de segmentos TCP é particularmente delicada.
 - (a) Indique os inconvenientes de um alarme cuja duração é demasiado curta.
 - (b) Indique os inconvenientes de um alarme cuja duração é demasiado longa.
 - (c) Indique resumidamente como é que no protocolo TCP a duração do alarme usado pelo emissor é calculado.
15. Considere uma conexão TCP a funcionar num contexto em que o RTT é de 100 ms e o débito máximo possível entre os extremos é de 100 Mbps. Admita que não pode usar a opção *Window Scaling Factor*.
 - (a) Qual o débito máximo teórico que essa conexão TCP pode atingir?
 - (b) O que faria para tentar melhorar o seu desempenho?
16. A implementação do protocolo TCP no servidor C é tal que uma conexão é fechada se esta não receber ou emitir segmentos durante 5 minutos. Invente uma forma de o TCP de um cliente de C evitar esse fecho, mesmo que os programas que estão a usar a conexão não necessitem de trocar dados durante 5 minutos. A sua solução não pode violar os requisitos de fiabilidade do protocolo TCP e deve ser transparente para as aplicações nos extremos da conexão.
17. Durante o funcionamento de uma conexão TCP todos os segmentos contêm um campo de nome *Receiver Advertised Window* que é sempre válido (pois não há nenhuma *flag* e indicar a sua validade ou não). É possível esse campo tomar o valor 0? Como é que o emissor que recebe um segmento com esse valor a 0 quebra a potencial situação de bloqueio em que pode ficar?
18. Considere uma conexão TCP entre dois computadores A e B com o MSS = 1024 bytes. O computador A transmite um ficheiro de 1.000 Kbytes para B. Admitindo que não há perda de pacotes na rede, estime a quantidade de segmentos transmitidos de B para A só com ACKs, e com ACKs e dados. Considere também os segmentos necessários para a abertura e fecho da conexão.
19. Na abertura de uma conexão TCP é fixado um valor de MSS.
 - (a) Admitindo que o MSS recomendado pelo nível rede das duas partes é maior que a dimensão do maior *frame* que cabe num dos canais atravessado pelos segmentos da conexão, o *Three-Way Handshake* inicial provoca obrigatoriamente fragmentação?
 - (b) Depois da conexão estabelecida, o MSS pode variar mais tarde quando se usa a versão 4 do protocolo IP?
 - (c) Depois da conexão estabelecida, o MSS pode variar mais tarde quando se usa a versão 6 do protocolo IP?
20. Durante a abertura de uma conexão TCP pretende-se avaliar o MSS a usar. Para isso, o lado que abre a conexão poderia começar por enviar um segmento TCP SYN em que a dimensão do segmento é artificialmente fixada como sendo a maior dimensão de *frame* da sua interface de ligação ao resto da rede. Descreva

como esse lado poderia usar o protocolo ICMP para descobrir o MSS correcto que deveria usar. Descreva também o que deve fazer a parte que aceita a conexão, com o mesmo objectivo, quando envia o segmento SYNACK de resposta.

21. Considere um ficheiro de dimensão D bytes a transferir de A para B através de TCP. Assuma que o MSS da conexão TCP a utilizar é 1000 bytes.
 - (a) Qual é a menor valor de D que obriga a que os números de sequência do TCP sejam reutilizados? Para efeito dos seus cálculos admita que $2^{10} \approx 10^3$, $2^{20} \approx 10^6$ e $2^{30} \approx 10^9$.
 - (b) Quantos segundos leva a transmitir o ficheiro com a dimensão calculada na alínea anterior, admitindo que o emissor e o receptor têm *buffers* infinitos, que não há perda de pacotes, que o RTT é 10 ms, que o débito máximo de extremo a extremo é de 100 Mbps, e que não pode usar a opção *Window Scaling*.
22. Indique porque razão o protocolo TCP, em particular a usar a opção SACK, não pode ter uma janela de emissão maior que o maior intervalo de números de sequência a dividir por 2, ou seja, 2^{31} .
23. Indique porque razão o mecanismo FAST RETRANSMIT do protocolo TCP desencadeia o procedimento GBN após três ACKs repetidos e não dois.
24. Considere uma conexão TCP entre os processos A e B ligados através da Internet. O módulo TCP do lado do processo A, que abriu a conexão e foi o primeiro a fechá-la, está no estado FIN_WAIT_2 e a seguir passou ao estado TIME_WAIT, onde se manteve por mais algumas dezenas de segundos. Porquê?

Capítulo 8

Controlo da saturação da rede

The scientist described what is: the engineer creates what never was.

– Autor: *Theodor von Karman – the father of the supersonic flight*

Como todos os sistemas construídos pelo homem, uma rede de computadores tem um capacidade limitada e um nível de utilização em que o seu rendimento é máximo. Muitos sistemas dispõem de mecanismos de *feedback* que permitem aos utilizadores procurarem o nível de solicitação ideal para a sua capacidade e para maximizarem o seu rendimento. Por exemplo, a maioria dos automóveis modernos dispõem de indicadores da velocidade e de consumo instantâneo que permitem ao condutor ajustar a sua condução a diferentes relações velocidade / consumo da viatura. Será fácil fazer algo de semelhante numa rede?

Numa rede de computadores em que a soma das capacidades dos canais de acesso dos computadores seja superior à capacidade de encaminhamento efectivo de pacotes pela rede, que é a situação mais frequente, é possível ao conjunto dos computadores injectarem pacotes a um débito superior ao que a rede é capaz de encaminhar. Numa rede IP essa situação irá traduzir-se na supressão de pacotes pelos comutadores. No entanto, a verdade é que nada impede um computador de aumentar o ritmo com que envia pacotes até atingir o débito máximo do canal que o liga à rede. Mas, se o fizer, está a melhorar o seu desempenho? A resposta depende do caminho tomado pelos seus pacotes e da actividade dos outros computadores com os quais está em competição.

Na grande maioria das situações, o aumento do débito de emissão dos computadores ligados a uma rede IP para além do que esta suporta, torna-se negativo para todos os computadores, incluindo os mais “gulosos”, e o respectivo débito de extremo a extremo até pode decrescer.

Idealmente, os computadores deveriam tentar não fazer solicitações para além do nível em que o rendimento da rede atinge o seu máximo. Infelizmente, em quase todas as redes, os comutadores no interior da rede ao detectarem a situação de saturação não têm capacidade de refrear os computadores.

É possível estabelecer uma analogia directa com o tráfego viário. Quando o ritmo dos automóveis que entram numa rede viária é superior ao que esta é capaz de acomodar, o resultado são engarrafamentos como os da Figura 8.1. Actualmente os sistemas de controlo do tráfego urbano até são capazes de detectar essas situações (*e.g.*, através de câmaras) mas de facto não têm muita capacidade para impedir os automóveis de

tentarem entrar nas zonas engarrafadas e agravarem a situação.¹



Figura 8.1: Um engarrafamento de tráfego automóvel devido à chegada de mais veículos por unidade de tempo do que aquele que essa zona da rede viária é capaz de acomodar

Nas primeiras redes de circuitos cada cliente só podia utilizar a capacidade reservada para o seu circuito, e um circuito só era estabelecido se a rede tivesse capacidade disponível para o acomodar. O problema da saturação² era resolvido negando a entrada de mais clientes, mesmo que houvesse capacidade para os acomodar tendo em consideração o consumo real dos circuitos activos. No outro extremo, as redes IP iniciais não impunham nenhum limite, voluntário ou involuntário, ao tráfego injectado na rede. O resultado foi catastrófico, e com a subida da sua popularidade, a Internet em geral exibia níveis de saturação muito elevados e detectavam-se quebras bruscas de débito extremo a extremo de 100 para 1. O comportamento era frequentemente o de uma rede viária em colapso pois nada progredia significativamente.

Esse problema foi resolvido através da introdução de mecanismos de controlo da saturação.

Os mecanismos de controlo da saturação (*congestion control mechanisms*) são mecanismos que procuram controlar *dinamicamente* as solicitações feitas à rede, de forma a maximizar o rendimento da mesma, tendo em consideração a utilização real e a capacidade disponível.

Os mecanismos de controlo da saturação serão tanto mais adequados quanto menos realizarem reservas *a priori*, quanto mais promoverem a equidade, e quanto mais permitirem um melhor aproveitamento de toda a capacidade disponível.

¹Existem, no entanto, planos para conseguir alterar automaticamente os trajectos de automóveis sem condutor em função do estado da rede viária.

²O termo usado em inglês é *congestion*, que pode ser traduzido por “saturação”, “congestão” ou “engarrafamento”. Geralmente, “congestão” é usado como sinónimo de afluência anormal de sangue a um órgão e “engarrafamento” como afluência anormal de veículos. O termo saturação é o mais transversal a diversos campos e por isso preferimos usá-lo.

8.1 Em que consiste a saturação

O desempenho das redes de computadores é estudado formalmente usando a teoria das filas de espera. À luz desta teoria o desempenho de um serviço é caracterizado pela taxa de trabalho produzido por unidade de tempo. A letra que se costuma usar para denotar essa grandeza é a letra grega λ (lambda). Aplicada a uma rede de computadores, o trabalho produzido é medido em unidades de informação entregues ao destino por unidade de tempo. Se todos os pacotes fossem da mesma dimensão, pode-se usar a unidade pacotes por segundo (*pps*). No caso mais geral deve-se usar a unidade bits por segundo (*bps*).

Se olharmos para a rede como uma caixa preta, ver a Figura 8.2, num determinado período de observação os computadores injectam na rede uma certa quantidade de pacotes, da qual resulta o tráfego médio injectado na rede $T_{in} = \sum \lambda_{in}(i)$, onde $\lambda_{in}(i)$ denota a informação injectada por unidade de tempo em cada canal de entrada. No mesmo período a rede entrega aos computadores que lhe estão ligados o tráfego médio $T_{out} = \sum \lambda_{out}(i)$, onde $\lambda_{out}(i)$ denota a informação entregue ao destino por unidade de tempo através de cada canal de saída.

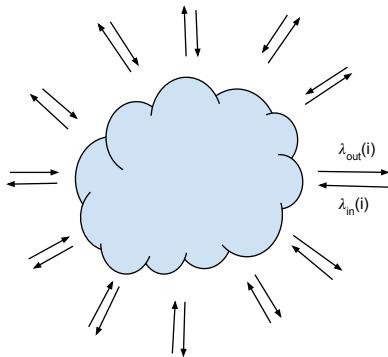
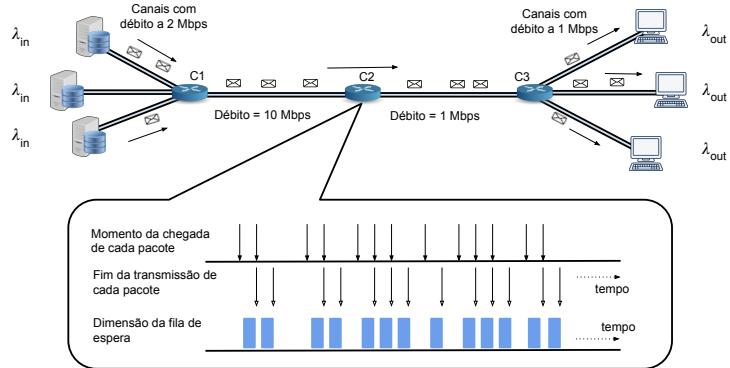


Figura 8.2: Tráfego injectado e tráfego entregue num determinado período

Em nenhum momento $T_{out} > T_{in}$ pois, por hipótese³, a rede não gera ela própria pacotes. Se tudo estiver a funcionar bem $T_{out} \approx T_{in}$ pois alguns pacotes perdem-se por erros nos canais. Quando a rede se aproxima da saturação T_{out} começa a ser significativamente inferior a T_{in} . Numa rede completamente saturada, existe um grande desperdício e $T_{out} \ll T_{in}$. Vamos a seguir ver porquê.

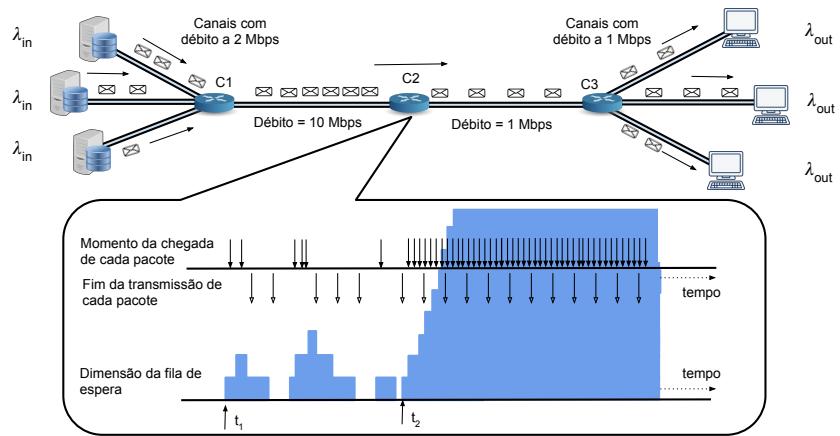
Para esse efeito vamos usar um modelo de rede simples como o da Figura 8.3. Os computadores à esquerda, os servidores, só injectam tráfego na rede, e este é sempre dirigido aos computadores da direita, os clientes. Todos os canais são ponto-a-ponto, bidireccionais e *full-duplex*. Os dois canais que ligam os três comutadores entre si, modelizam uma rede de trânsito simplificada. Os canais que ligam os computadores a um comutador modelizam duas redes de acesso, uma à esquerda e outra à direita. O débito agregado dos canais de acesso dos servidores (3×2 Mbps) é inferior ao do canal que liga os comutadores C1 a C2, que tem o débito de 10 Mbps. O débito agregado dos canais de acesso dos clientes (3×1 Mbps) é superior ao do canal de trânsito que liga os comutadores C2 a C3, com o débito de 1 Mbps e que é o canal de gargalo (*bottleneck link*) da rede. Admite-se que não existem erros nos canais

³ O tráfego *multicast* e *broadcast* viola esta hipótese pois a injeção de um único pacote, pode conduzir à geração de muitos outros.

Figura 8.3: Filas de espera no interior da rede com $T_{out} = T_{in}$

pelo que estes não perdem pacotes. Por hipótese também, caso os emissores usem um protocolo que emita ACKs, o tempo de transmissão desses pacotes é desprezável e a rede encaminha-os sem problemas dos clientes para os servidores. Finalmente, considera-se igualmente que é realista desprezar o espaço ocupado pelos cabeçalhos e considera-se que os pacotes de dados só transportam informação útil.

A Figura 8.3 mostra igualmente a chegada de pacotes ao comutador C2 e o estado da fila de espera do seu canal de saída. Na situação ilustrada na figura, quando chega um novo pacote, este encontra sempre a fila de espera vazia, isto é, todos os pacotes chegados anteriormente já foram transmitidos. Nesta situação a fila de espera só tem no máximo um pacote, exactamente o pacote que está a ser transmitido, e o atraso máximo introduzido pelo comutador C2 é o tempo de transmissão do pacote em trânsito. A rede está muito folgada, com pouca carga e encaminha sem problemas até ao destino os pacotes que nela entram e portanto $T_{out} = T_{in}$.

Figura 8.4: Filas de espera no interior da rede com $T_{out} = T_{in}$ de t_1 a t_2 e $T_{out} > T_{in}$ a partir de t_2

A Figura 8.4 mostra uma situação em que os servidores começaram a aumentar o ritmo com que injectam pacotes dirigidos aos clientes. Entre os momentos t_1 e t_2 alguns pacotes chegam em grupos ao comutador C2, e a fila de espera de espera cresce momentaneamente, mas é sempre possível transmitir cada grupo de pacotes antes que chegue o grupo seguinte. Nesse período continua a verificar-se $T_{out} = T_{in}$ mas alguns pacotes começam a ser encaminhados com um tempo de trânsito extremo a extremo variável e maior que anteriormente (*i.e.*, aparece *jitter*).

No entanto, a partir do momento t_2 , chega um grande grupo de pacotes contíguos e a fila de espera começa a crescer significativamente. Admitindo que essa fila de espera é infinita, mesmo que a chegada ininterrupta de pacotes continue, a rede vai continuar a entregar todos pacotes, mas estará limitada ao débito do canal gargalo (*bottleneck link*). Seja qual for a taxa com que chegam pacotes ao comutador, este só entrega pacotes com o débito de 1 Mbps e nessa altura $T_{in} > T_{out} = 1 \text{ Mbps}$. Naturalmente, a fila de espera vai crescendo sem limite. Admite-se igualmente que o protocolo usado pelos emissores é ingênuo e continua a emitir pacotes seja qual for a situação na fila de espera e o tempo de trânsito extremo a extremo experimentado pelos pacotes.

Define-se débito útil de informação ou *goodput* como sendo o débito útil de informação entregue pela rede aos destinatários, isto é, descontando todo o desperdício introduzido pelos protocolos, quer em termos de cabeçalhos dos pacotes, quer em termos de pacotes reemittidos e recebidos em duplicado (neste modelo, como já referimos, desprezam-se os cabeçalhos dos pacotes).

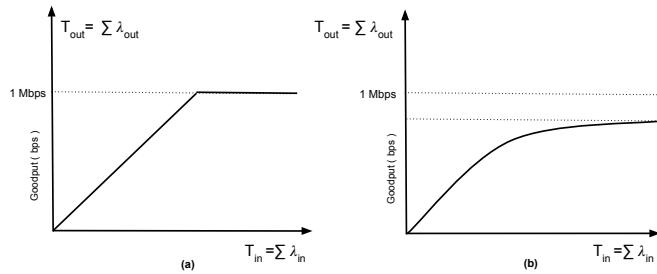


Figura 8.5: Débito útil (*goodput*) com filas de espera infinitas: (a) sem retransmissão de pacotes pelos emissores, (b) com retransmissão prematura de pacotes

A Figura 8.5 mostra a relação do *goodput* com T_{in} em duas situações diferentes, correspondentes a dois comportamentos distintos dos emissores. Em ambas as situações, sempre que $T_{in} < 1\text{Mbps}$, a rede está pouco carregada e verifica-se que $T_{out} = goodput = T_{in}$. No entanto, no caso do gráfico (a), sempre que $T_{in} > 1\text{Mbps}$, *goodput* = 1Mbps pois esse é o limite da capacidade de encaminhamento da rede de trânsito, mas a fila de espera de C2 vai crescendo e o tempo de trânsito extremo a extremo também. O gráfico (a) corresponde à situação acima referida, em que a fila de espera do comutador é infinita e os emissores usam um protocolo simplista e transmitem logo que podem os pacotes das aplicações.

No entanto, os protocolos de transmissão fiável de dados normais usam ACKs e são sensíveis ao tempo de trânsito dos pacotes. Mas como a fila de espera é infinita e os canais não perdem pacotes, todos os alarmes que dispararem são necessariamente prematuros. Assim, quanto maior for a fila de espera de C2, mais os pacotes tenderão a ficar na fila de espera, mais alarmes prematuros irão disparar nos emissores e mais pacotes duplicados serão injectados na rede. Resultado, a verdadeira relação entre T_{in} e o *goodput* é a que é mostrada pela Figura 8.5 através do gráfico (b) pois todos os pacotes emitidos em duplicado consomem inutilmente uma parte da capacidade

disponível.

De qualquer forma, o gráfico (b) assume que o valor dos *timeouts* vai aumentando dinamicamente para acomodar o aumentado do tempo de trânsito extremo a extremo, pois pressupõe-se, tal como no caso do TCP, que o valor dos *timeouts* é ajustado em função do RTT estimado. Se o valor do *timeout* fosse constante, como a fila de espera vai crescendo infinitamente, cada vez disparariam mais alarmes, cada vez mais pacotes seriam retransmitidos prematuramente, e a fracção de novos pacotes úteis recebidos pelos receptores sobre o número total de pacotes que cruzariam a rede iria descer, e o *goodput* também, dado que todos os pacotes, incluindo os duplicados, partilham a capacidade do canal gargalo ou *bottleneck link*.

Isto mostra mais uma vez que entregar pacotes muito atrasados é contraproducente e que portanto mais vale suprimir os pacotes que de qualquer forma vão chegar muito para lá do razoável num determinado contexto. Um modelo de rede mais realista é aquele em que as filas de espera são limitadas, e os pacotes que não têm lugar nas filas de espera são suprimidos como mostra a Figura 8.6. Isso também assegura que o tempo de trânsito extremo a extremo passará a ser limitado pelo somatório do tempo de transmissão das filas de espera máximas de cada um dos comutadores atravessados.

Adicionalmente, a rede da Figura 8.6 tem uma configuração mais complexa, pois no comutador C3, ligado directamente aos clientes, convergem agora canais vindos de outros comutadores que injectam outros pacotes em competição com os que os servidores enviam. Isso quer dizer que alguns dos pacotes que consumiram a capacidade do canal que liga C2 a C3 vão ser suprimidos devido a essa competição, e que uma cada vez maior fracção da capacidade desse canal não contribuirá para o débito útil final.

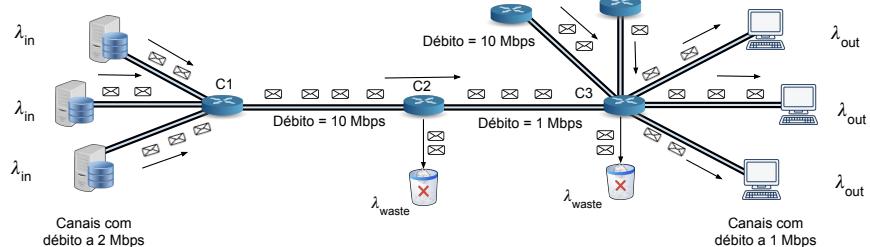


Figura 8.6: Modelo de rede com filas de espera finitas e supressão de pacotes

Como mostra a Figura 8.7, ao aumentar o tráfego injectado na rede (T_{in}), a fracção desse tráfego que atinge o destino e contribui para o débito útil da rede vai decaindo devido ao efeito combinado da existência de canais que estão saturados e limitam a capacidade da rede, da cada vez maior retransmissão prematura de pacotes atrasados e da supressão de pacotes que já consumiram capacidade na rede.

Assim, torna-se claro que se o conjunto do tráfego injectado na rede for tal que a rede funcione abaixo da sua zona de *stress*, o seu rendimento será máximo, quer pelo número reduzido de pacotes que se perdem, quer pelo facto de não haver pacotes que consomem recursos e aumentam o tempo de trânsito mas não chegam ao destino. A Figura 8.8 põe em evidência que se a rede funcionar abaixo da zona sombreada, a sua zona de *stress* ou de saturação, os pacotes chegam ao destino e o tempo de trânsito de extremo a extremo é próximo do seu mínimo, sem dilatações artificiais devidas a longas filas de espera.

O rendimento da rede tem sido “medido” informalmente nos parágrafos anteriores em termos da quantidade de pacotes entregues no destino, desprezando a faceta tempo de trânsito. Para captar as duas facetas foi introduzida a noção de **poder da rede**

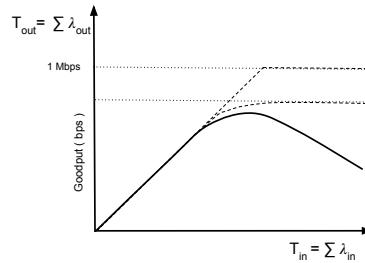


Figura 8.7: *Goodput* com filas de espera finitas, retransmissão prematura de pacotes atrasados e supressão de pacotes que consumiram recursos no *bottleneck link* antes de serem suprimidos

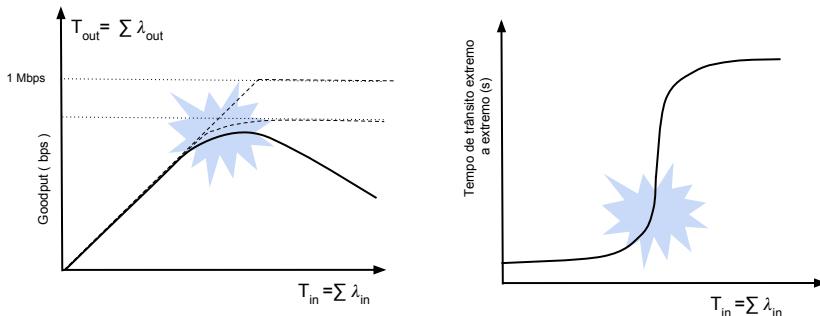


Figura 8.8: *Goodput* e tempo de trânsito numa rede saturada

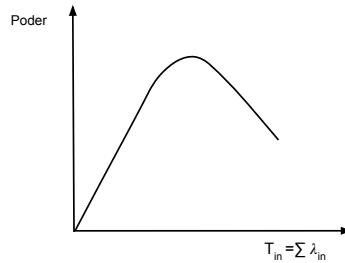
(*network power*) [Kleinrock, 1979] que é definido pela equação:

$$\text{Power} = \frac{\text{goodput}}{\text{delay}} \quad (8.1)$$

O gráfico da Figura 8.9 representa o poder de uma rede como o do modelo usado na discussão. À medida que o tráfego injectado na rede e o débito útil sobem, o poder da rede vai crescendo pois o tempo de trânsito extremo a extremo é sensivelmente constante. O poder atinge o seu ponto máximo quando a rede começar a ficar saturada e o *goodput* começar a decrescer. Ultrapassado esse ponto, o *goodput* decresce e o tempo de trânsito extremo a extremo aumenta, provocando a queda significativa do poder da rede, visto que a mesma está agora completamente saturada.

Por essa razão, é importante encontrar formas de conseguir que o tráfego injectado na rede nunca ultrapasse a zona em que o poder atinge o seu máximo. Na verdade é preferível até trabalhar sempre numa zona com alguma folga pois, na prática, os fluxos de tráfego são irregulares e contém picos de débito repentinos que podem momentaneamente induzir saturação. Para se conseguir atingir este objectivo são usadas arquitecturas e protocolos que implementam algoritmos de controlo da saturação.

Uma rede entra em saturação quando o conjunto das solicitações de encaminhamento de pacotes que recebe conduzem a uma situação de colapso, e a quantidade real

Figura 8.9: Poder da rede em função de T_{in}

de pacotes encaminhados é inferior à capacidade efectiva da rede.

Define-se **débito útil (goodput)** como sendo o débito útil de informação entregue pela rede aos destinatários, isto é, descontando todo o desperdício introduzido pelos protocolos, quer em termos de cabeçalhos dos pacotes, quer em termos de pacotes reemitidos e recebidos em duplicado.

Define-se **poder da rede (network power)** como sendo o quociente do débito útil pelo tempo de trânsito de extremo a extremo. Numa rede saturada, o aumento do débito de injecção de pacotes na rede conduz à diminuição do seu poder.

O conjunto de arquitecturas, protocolos e algoritmos usados para controlar as solicitações à rede, de forma a evitar que esta entre em saturação, chamam-se **arquitecturas, protocolos e algoritmos de controlo da saturação (congestion control architectures, protocols and algorithms)**.

8.2 Como controlar a saturação

Um computador ligado à rede injecta tráfego dirigido a diferentes destinos que, naturalmente, atravessa zonas distintas da rede. Por isso o controlo do débito não pode ser feito por computador, mas pode ser feito por fluxo de pacotes, por exemplo por fluxo de transporte, *i.e.*, um conjunto de pacotes de dados com origem e destino nos mesmos sockets (*e.g.*, por conexão TCP, por sequência de datagramas UDP com a mesma origem e dirigidos ao mesmo destino, *etc.*).

No entanto, procurar tirar o melhor rendimento possível da rede para maximizar o seu poder não é fácil, pois existem diversos factores que tornam o problema complicado, nomeadamente os seguintes:

1. Os fluxos que atravessam a rede são dinâmicos e, no caso geral, alteram-se continuamente. A duração, os requisitos e a origem e destino dos fluxos existentes numa rede geral exibem uma grande variedade e uma grande escala. O controlo da saturação **tem de ser dinâmico** e adaptar-se continuamente ao conjunto dos fluxos que atravessam a rede em cada momento. A **convergência** dos algoritmos e protocolos de controlo da saturação mede o tempo que é necessário para adaptar a gestão dos fluxos na sequência de uma alteração das solicitações

à rede. Se a convergência é lenta, a gestão dos fluxos leva demasiado tempo a adaptar-se às mudanças, o que é sub-óptimo. Se for demasiado rápida, a situação pode oscilar e mesmo assim aparecer saturação.

2. Na grande maioria das situações, os computadores são autónomos e independentes da rede, pois pertencem a entidades distintas e com objectivos parcialmente contraditórios (*e.g.*, clientes e fornecedores). Por esta razão **não é possível usar arquitecturas de controlo da saturação baseadas em integração e coordenação** elevadas entre a gestão da rede e os algoritmos e protocolos que se executam nos computadores. A excepção a esta regra são as redes internas aos centros de dados em que a gestão da rede e dos servidores do centro pode ser feita de forma integrada.
3. Os diferentes fluxos que atravessam a rede têm origens e destinos distintos e seguem por diversos caminhos, por isso encontram **diferentes canais gargalos**. Encontrar o débito adequado a cada fluxo depende também dos fluxos que com ele se cruzam e competem.
4. Numa rede sem diferenciação da qualidade de serviço fornecida aos diferentes clientes, os fluxos têm de obter uma fração equitativa da capacidade da rede. No entanto, essa equidade não é simples de estabelecer e não basta dividir igualmente a capacidade da rede pelos diferentes fluxos. Na verdade, **a noção de divisão equitativa só é fácil de aplicar localmente**, *i.e.*, ao conjunto dos fluxos que partilham o mesmo canal gargalo (*bottleneck link*).

Os dois primeiros pontos acima introduzem as dificuldades ligadas ao dinamismo do problema, e à dificuldade de integrar a gestão da rede e a gestão dos fluxos gerados pelos computadores na maioria dos casos. Os dois últimos pontos estão relacionados com o problema da afectação da capacidade disponível aos diferentes fluxos e com a necessidade de realizá-lo de forma equitativa.

Para pôr em evidência os contornos das duas últimas questões (ignorando os problemas do dinamismo e de encontrar uma implementação viável), vamos usar um algoritmo centralizado para afectar estaticamente o débito máximo a cada um dos fluxos presentes na rede.

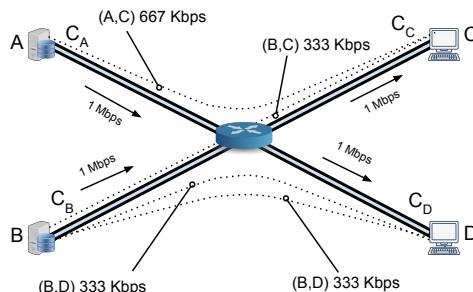


Figura 8.10: Determinação do débito máximo dos fluxos (A,C), (B,C) e 2×(B,D) numa rede em estrela

O algoritmo requer o conhecimento prévio da configuração da rede, do conjunto dos fluxos (cada fluxo é caracterizado pelo débito solicitado e pelo par de nós origem e destino) e do caminho que cada um deles segue na rede. O resultado do algoritmo é o débito máximo de cada um desses fluxos que maximiza o *goodput* e minimiza a perda de pacotes e o tempo de trânsito extremo a extremo (*i.e.*, maximiza o poder da rede).

O algoritmo usa uma pequena quantidade de tráfego designado *quantum*, denotado λ_δ , e consiste em ir incrementando de forma rotativa cada fluxo com esse *quantum*⁴. Logo que um fluxo esgota a capacidade do respectivo canal gargalo, deixa de ser incrementado. O algoritmo termina quando não for possível incrementar o débito de mais nenhum fluxo e o resultado final é o débito que foi afectado a cada fluxo.

Por exemplo, dada a rede da Figura 8.10, constituída por 4 computadores A, B, C e D interligados por um comutador através de canais com o débito de 1000 Kbps e um tempo de propagação desprezável, pretende-se encaminhar os quatro fluxos (A,C), (B,C) e 2× (B,D) afectando a cada um o máximo que for possível. Aplicando o algoritmo usando $\lambda_\delta = 1$ Kbps, chega-se à seguinte afectação: 667 Kbps a (A,C) e 333 Kbps a todos os outros, pois os restantes 3 fluxos têm o canal C_B como canal gargalo. Repare-se também que os canais C_A e C_D não são saturados e que o canal C_C é partilhado em 2/3 pelo fluxo (A,C) e 1/3 pelo fluxo (B,C).

Cada fluxo só obtém a capacidade correspondente à divisão, de forma aproximadamente equitativa, da capacidade do seu canal gargalo como acontece no caso dos três fluxos com origem em B. No entanto, no caso do fluxo com origem em A, o seu canal gargalo é o canal C_C , partilhado com outro fluxo que, como tem um canal gargalo diferente, deixa cerca de 2/3 da capacidade livre para o seu competidor.

Este tipo de afectação equitativa chama-se **max-min fairness** [Boudec, 2014] e é caracterizada por não ser possível incrementar a capacidade afectada a nenhum fluxo, sem decrementar a capacidade afectada a outros fluxos que já estão a receber uma fração igual ou inferior à que esse já recebe.

Definir uma arquitectura e um conjunto de protocolos que usasse um algoritmo centralizado para determinar a capacidade máxima de cada fluxo é bastante difícil no caso geral, pois isso exige conhecer todos os fluxos existentes na rede, qual o caminho que estes seguem, qual o estado exacto da rede, correr o algoritmo em tempo útil e ainda conseguir comunicar aos computadores as afectações determinadas para cada um dos seus fluxos. Tudo isto realizado dinamicamente e a uma escala de milhares ou mesmo milhões de fluxos.⁵

Tirar o melhor rendimento possível da rede, sem a saturar e assegurando equidade entre os diferentes fluxos de pacotes é um problema de optimização.

Calcular a solução óptima implica dispor de uma visão global do estado da rede, dinamicamente actualizada em função da evolução das solicitações e, adicionalmente, é necessário conseguir controlar os computadores ligados à rede para controlar o ritmo de emissão de pacotes por cada um dos seus fluxos.

Este tipo de arquitectura não é realista senão numa situação em que os gestores da rede tivessem controlo completo sobre os computadores que lhe estão ligados. No caso geral, o máximo que a rede pode esperar é alguma colaboração dos protocolos de transporte que esses computadores usam.

Para introduzir controlo da saturação, existem diversas alternativas de solução que se traduzem em diferentes algoritmos e protocolos assim como em diferentes arquitecturas de coordenação da rede e dos computadores. A figura 8.11 ilustra três grupos de alternativas diferentes.

⁴ Como em muitos algoritmos deste tipo usa-se o termo *quantum* para designar uma pequena quantidade cujo valor determina o erro cometido nos cálculos. O valor usado na prática é o maior valor que conduz a um erro menosprezável num dado contexto.

⁵ Acresce, no caso mais geral, que a Internet é um conjunto de redes interligadas com gestão autónoma e esta afectação só poderia ser realizada por um mecanismo que coordenasse o conjunto de todas as redes.

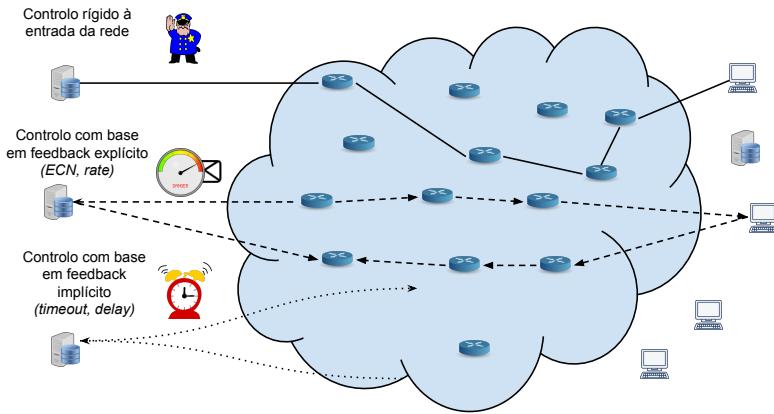


Figura 8.11: Alternativas para lidar com a saturação da rede

Começando de cima para baixo, o primeiro grupo de soluções baseia-se numa integração muito grande entre os computadores e a rede. Por exemplo, numa rede baseada na noção de circuito, um fluxo e um circuito podem coincidir e um computador só pode dar início a um fluxo depois de a rede ter aberto o respectivo circuito e lhe ter afectado a capacidade solicitada. Se o circuito não “couber” na rede, a rede recusa-o. Mesmo sem usar necessariamente a noção de circuito, este tipo de arquitectura de controlo da saturação inclui sempre mecanismos de reserva de recursos e de limitação do débito máximo que os computadores podem injectar na rede. Se esses limites forem conservadores anulam a possibilidade de haver saturação à custa de uma arquitectura rígida e que subaproveita a capacidade da rede, pois desperdiça a capacidade que não estiver a ser usada no momento. Adicionalmente, esta solução é pouco escalável e adapta-se com dificuldade a um grande dinamismo.

Os outros dois grupos de soluções de controlo da saturação não impõem limites *a priori*, nem ao número, nem ao débito dos fluxos. Adoptam um ponto de vista optimista, baseado na noção de “melhor esforço”, que não anula *a priori* a saturação, pois esta pode sempre ter lugar. Em contrapartida, usam mecanismos de *feedback* para indicarem ao nível de transporte dos computadores que deve reduzir o débito dos fluxos, sob pena de introduzir saturação. Baseiam-se portanto na colaboração e na disponibilidade voluntária do nível transporte para reduzir o débito de emissão de pacotes.

No primeiro grupo deste tipo de soluções esse *feedback* é explícito e pode assumir diversas formas: a rede sinaliza um fluxo de que o mesmo está em risco de provocar saturação e que portanto é preferível reduzir o seu débito (solução ECN – *Explicit Congestion Notification*) ou então é comunicado o débito explícito a adoptar pelo fluxo (*Explicit Rate Notification*).

No segundo grupo utiliza-se *feedback* implícito, o qual se traduz em o nível de transporte dos computadores medir o comportamento dos seus fluxos para detectar a presença de saturação (perda de pacotes e variação do débito e do tempo de trânsito extremo a extremo). Neste caso a rede adopta uma posição do tipo *laissez faire laissez passer* e espera que os seus utilizadores adoptem uma melhor postura e, ao aperceberem-se de que esta está saturada, reduzam voluntariamente o débito dos seus

fluxos. Em qualquer das situações, se as filas de espera estiverem demasiado cheias, são suprimidos pacotes.

O controlo da saturação necessita de regulação do débito dos fluxos de pacotes emitidos pelos computadores ligados à rede. Este controlo pode basear-se em imposição pela rede do débito máximo ou em *feedback* da rede aos computadores para que estes se auto-regulem.

As arquitecturas baseadas em *feedback* da rede dividem-se ainda em *feedback* explícito, quando a rede dá indicações directas aos computadores para que estes se auto-regulem, e *feedback* implícito, quando são os computadores que detectam os indícios de saturação.

O protocolo TCP incorpora algoritmos de controlo da saturação baseados em *feedback* implícito e, opcionalmente, também explícito (a solução ECN). Assim vamos estudar implementações possíveis destas duas alternativas através do estudo do controlo da saturação no protocolo TCP.

8.3 Controlo da saturação no protocolo TCP

Os protocolos de janela deslizante proporcionam uma forma simples de controlo do ritmo de emissão: a dimensão da janela de emissão. De facto, uma conexão com janela de emissão de dimensão W transmite dados com o débito médio de W/RTT , ver a Secção 6.3.

O protocolo TCP usa a dimensão da janela de emissão para concretizar quer um mecanismo de controlo de fluxo, que adapta o ritmo de transmissão à capacidade do receptor consumir atempadamente os dados recebidos, quer um mecanismo de controlo da saturação, que adapta o ritmo de transmissão à capacidade disponível na rede para a conexão TCP.

Repare-se que a dimensão da janela de emissão do TCP limita o número máximo de segmentos que podem ser emitidos em sequência mas, uma vez todos os segmentos da janela transmitidos, os seguintes são transmitidos exactamente ao ritmo da chegada dos ACKs. Ora os ACKs vêm ao ritmo com que o receptor vai recebendo de facto segmentos, o qual é condicionado pela capacidade do canal com menos capacidade, o canal gargalo ou *bottleneck link*.

Por exemplo, ver a Figura 8.12, considere-se um emissor e um receptor ligados ao mesmo comutador. O canal do emissor tem um débito de 1 Gbps, e o do receptor tem um débito de 100 Kbps. Se a janela de emissão for igual a 4 segmentos de 1250 bytes cada (≈ 10.000 bits em cada segmento), o tempo de transmissão de cada segmento a 1 Gbps é negligenciável ($10^4/10^9 = 10 \mu s$). No entanto, o tempo de transmissão de cada segmento a 100 Kbps é 100 ms. Inicialmente, o emissor pode enviar os 4 segmentos em sequência em tempo negligenciável, mas os segmentos vão ser transmitidos pelo comutador através do canal do receptor, e este vai receber cada um deles espaçado do tempo de transmissão nesse canal (100 ms). Portanto, os ACKs são recebidos pelo emissor separados de pelo menos 100 ms e, após a sequência inicial, o emissor vai transmitir os outros segmentos separados de 100 ms cada, o ritmo com que recebe os ACKs.

O TCP é **auto regulado (*self-clocking*)** pois o ritmo de chegada de ACKs depende do ritmo com que os segmentos TCP são entregues ao receptor. Com efeito, sempre que um ACK é recebido, isso quer dizer que um segmento deixou a rede e outro pode ser transmitido.

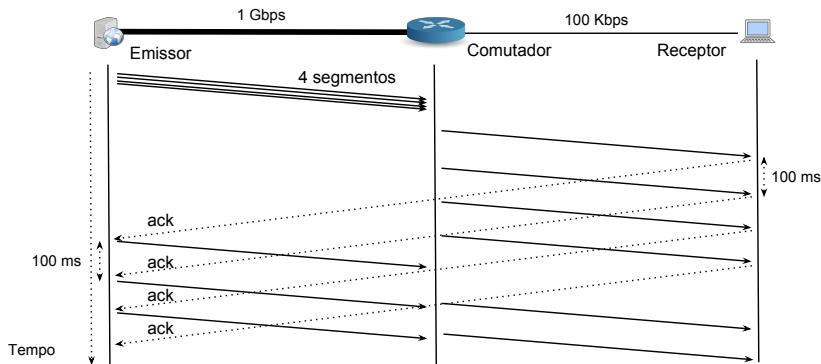


Figura 8.12: O débito do canal gargalo impõe o ritmo de chegada de ACKs e também o ritmo de envio de segmentos pelo emissor

Teoricamente, se apenas for limitada pelo débito disponível na rede, *i.e.*, não existem limitações aplicacionais nem de controlo de fluxos, uma conexão TCP poderia ajustar a dimensão da sua janela ao estritamente necessário para manter o débito permitido pelo canal gargalo e manter-se numa situação estável e ótima. Esta situação particular é muito difícil de obter porque a capacidade disponível no canal gargalo varia dinamicamente devido ao restante tráfego em competição e o TCP tem dificuldade em estimá-la. Uma forma de conseguir fazer esta estimativa consiste em ir aumentando progressivamente a janela, diminuindo-a apenas quando se apercebe que se passou o limite adequado.

De que sinais dispõe o TCP para se aperceber de que o fluxo de pacotes enviados tem um ritmo que não é suportado pelo canal gargalo? Existem dois tipos de sinal: a eventual perda de pacotes, sinalizada pelo disparo de alarmes (*timeouts*), e o tipo de variação experimentado pelo débito extremo a extremo e pelo tempo de trânsito dos pacotes.

Com efeito, se não existirem mecanismos de sinalização explícita pela rede da existência de saturação, o receptor não se apercebe da mesma pois desconhece o ritmo de emissão do emissor. Só o emissor, ao não receber atempadamente os ACKs, ou através da análise da variação RTT, poderá tentar inferir qual o estado da rede na zona que os seus pacotes atravessam, e em particular, qual o estado do canal gargalo que estes atravessam. O receptor também se apercebe de anomalias quando recebe pacotes fora de ordem, mas esse evento não está necessariamente ligado a saturação da rede.

O sinal transmitido pelo disparo de *timeouts* é não só o mais fiável, como também o mais simples de interpretar. Nas redes modernas, a perda de pacotes devido a erros nos canais é desprezável, excepto nos canais sem fios. Por outro lado, a forma como o TCP calcula a duração dos alarmes (ver a Secção 7.2), faz com que a retransmissão prematura de segmentos seja menos provável.

Como um emissor TCP não tem noção de qual a capacidade que tem disponível na rede terá de fazer experiências. Por exemplo, poderá começar com uma janela de dimensão igual à do MSS (*Maximum Segment Size*) e depois ir subindo ou descendo a sua dimensão. Se a variar lentamente, a convergência pode ser lenta e o emissor pode não aproveitar a capacidade disponível ou, em caso de saturação, pode levar demasiado tempo a corrigir o débito e a saturação persistir para além do desejável. Se a variação for brusca, o emissor poderá oscilar entre momentos de sub-aproveitamento

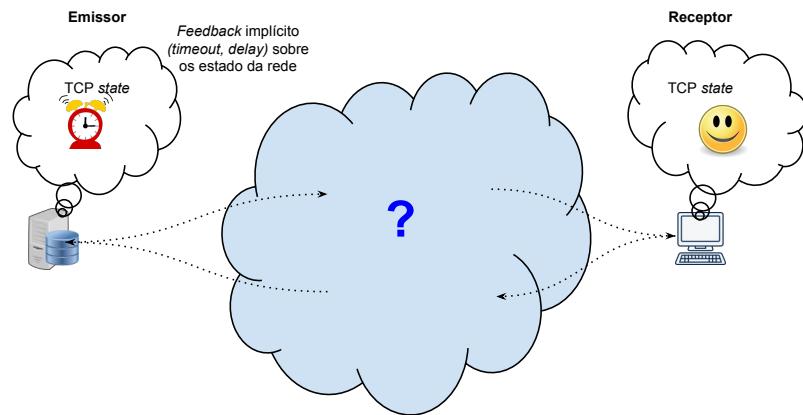


Figura 8.13: Controlo da saturação com base em *feedback* implícito

e momentos de saturação, sem encontrar um ponto de equilíbrio.

A variação é rápida se a janela de emissão mudar de forma multiplicativa (*e.g.*, sobe para o dobro ou diminui para metade em cada RTT) e será relativamente lenta (linear) se a mesma dimensão variar de forma aditiva (*e.g.*, sobe ou desce de 1 MSS em cada RTT). Resta saber, das quatro possíveis combinações de ritmos de variação (subida e descida aditivas, subida e descida multiplicativas, subida aditiva e descida multiplicativa, e vice-versa) qual será a mais adequada.

Como um emissor TCP não tem noção de qual o estado da rede, terá de experimentar ir aumentando o ritmo de emissão e, quando se aperceber que há problemas, deverá diminui-lo. O TCP varia o seu ritmo de emissão máximo usando a fórmula **subida aditiva, descida multiplicativa (AIMD – Additive Increase, Multiplicative Decrease)**.

A técnica de ir incrementando o ritmo de transmissão para tentar aproximar a capacidade máxima da rede sem a saturar, chama-se **sondar os recursos da rede (network resource probing)**.

A razão de ser desta opção, ver a Figura 8.14, pode ser justificada de forma intuitiva: a prudência manda não se ser demasiado abrupto a exigir recursos da rede, e quando se detectam anomalias é preferível recuar rapidamente para uma situação em que a saturação e a perda de pacotes sejam rapidamente ultrapassadas, tanto mais que nessa situação vários segmentos estarão ainda em trânsito. Com efeito, se o RTT de uma conexão for elevado e a saturação ocorrer perto do destino, existe uma elevada probabilidade de estarem muitos pacotes em trânsito. Veremos a seguir que é possível justificar mais rigorosamente que a estratégia AIMD é de facto a mais adequada, ver a Secção 8.5.

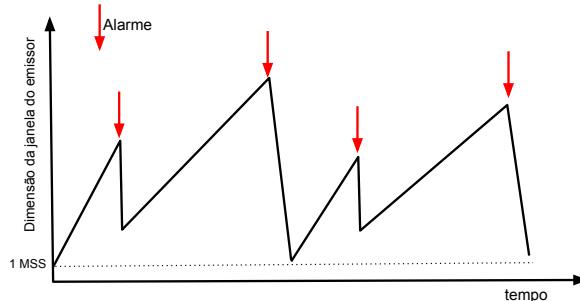


Figura 8.14: *Additive increase multiplicative decrease*

O protocolo TCP usa a dimensão da janela de emissão para concretizar quer um mecanismo de **controlo de fluxo (flow control)**, que adapta o ritmo de transmissão à capacidade do receptor consumir atempadamente os dados recebidos, quer um mecanismo de **controlo da saturação (congestion control)**, que adapta o ritmo de transmissão à capacidade disponível na rede para a conexão TCP.

O TCP ajusta dinamicamente a dimensão da janela de emissão de tal forma que essa dimensão nunca pode ser superior a dois limites. O primeiro limite designa-se *receiver advertised window (recAdvertisedWnd)*, e corresponde ao limite imposto pelo receptor através do mecanismo de controlo de fluxo. O segundo designa-se *congestion window (congWnd)*, e corresponde ao limite imposto pelo algoritmo de controlo da saturação.

O algoritmo de controlo da saturação do TCP tem início com `congWnd = mss`, incrementa o ritmo de transmissão subindo `congWnd`, usa o disparo do *timeout* como sinal da presença de saturação, da qual tenta fugir tão rapidamente quanto possível, e portanto reduz de novo o valor de `congWnd` ao valor do MSS na sequência de um *timeout*. Ou seja, a descida multiplicativa é a mais abrupta que é possível.

A subida aditiva é materializada através do incremento do valor de `congWnd` de MSS bytes em cada RTT. Para esse efeito, o TCP pode usar a seguinte estratégia: no início da conexão `congWnd` recebe o valor do MSS, após receber o primeiro ACK, *i.e.*, após um RTT, `congWnd` sobe para $2 \times \text{MSS}$, após mais 2 ACKs, outro RTT, sobe para $3 \times \text{MSS}$, após mais 3 ACKs, outro RTT, sobe para $4 \times \text{MSS}$, e assim sucessivamente. Pode-se incrementar o valor de `congWnd` de MSS em cada RTT ou incrementar ligeiramente o valor com cada ACK recebido. Com efeito, se o TCP estiver a usar uma janela limitada apenas pelo valor de `congWnd` e segmentos com a dimensão MSS, e se todos os segmentos forem *acked*, em cada RTT os ACKs recebidos serão tantos quanto o número de segmentos transmitidos, ou seja $\text{congWnd} / \text{MSS}$. A subida aditiva pode então ser implementada incrementando `congWnd` de $\text{MSS} \times \text{MSS} / \text{congWnd}$ cada vez que o TCP recebe um ACK, ver a Listagem 8.1, pois continua a admitir-se que em cada RTT se transferem `congWnd` bytes.

A estratégia de começar com uma janela igual a MSS foi chamada *slow start*, porque a janela do TCP começa sempre com um valor baixo, por oposição à solução usada

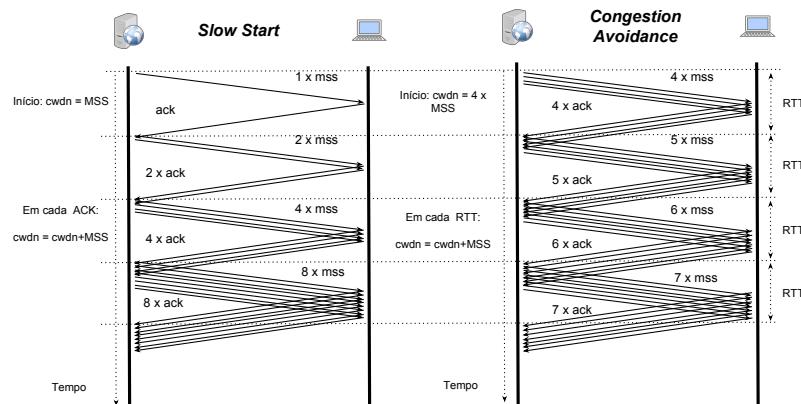
Listing 8.1: Pseudo código simplificado da actualização de `congWnd` com AIMD

```

start :      congWnd = MSS
alarm :      congWnd = MSS
ACK:        congWnd += MSS * MSS / congWnd

```

antes da introdução de controlo de saturação⁶. Por exemplo, supondo que se está numa situação em que o RTT é de 100 ms, o MSS de 1250 bytes (\approx 10.000 bits) e a capacidade disponível para a conexão de 1 Mbps, o débito inicial é $W/RTT = 10.000/0,1 = 100$ Kbps no 1º RTT, 200 Kbps no 2º RTT, 300 Kbps no 3º RTT, etc. e só no 10º RTT, ou seja ao fim de 1 segundo, é que se atinge o débito máximo disponível. Se o débito disponível for 10 Mbps, só ao fim de quase 10 segundos é que se atinge esse débito. A alternativa de usar um valor mais alto para a janela inicial é complicada pois o emissor não sabe com que situação real da rede está de facto a lidar.



O algoritmo concreto utilizado no protocolo TCP para realizar o controlo de saturação passou por várias fases. Inicialmente o TCP não tinha controlo da saturação mas, quando esta apareceu, Van Jacobson [Jacobson, 1988] liderou um notável trabalho de engenharia e experimentação que conduziu a duas versões iniciais do algoritmo, designadas respectivamente TCP Tahoe e TCP Reno.

Desde essa altura a Internet evoluiu imenso em termos de escala e capacidade, e durante esse período de quase 30 anos os algoritmos de controlo da saturação evoluíram muito e tiveram várias versões, mas muitas dessas novas versões correspondem a refinamentos e afinações cada vez mais sofisticadas destas duas versões iniciais. Por isso vamos dedicar-lhes alguma atenção a seguir.

A discussão que se segue assume que o débito do emissor é limitado apenas pelo algoritmo de controlo da saturação. A realidade é mais subtil pois em cada momento o TCP é limitado por vários factores: a aplicação ter dados para enviar ao ritmo adequado, o canal gargalo não ser o canal que liga o computador à rede e o receptor ter espaço nos *buffers* de recepção. De facto, a verdadeira janela de emissão é assim calculada:

Listing 8.2: Cálculo da dimensão da janela de emissão do TCP

```
maxWnd = min(congWnd, recAdvertisedWnd)
effectiveWnd = maxWnd - (lastByteSent - lastByteAcked)
if (effectiveWnd > 0)
    // it is possible to send more data ...
```

e em cada momento só é possível transmitir dados se `effectiveWnd > 0`. Ou seja, quando `maxWnd = MSS` na sequência de um *timeout*, não podem ser transmitidos novos dados enquanto não forem retransmitidos e/ou *acked* os segmentos em trânsito.

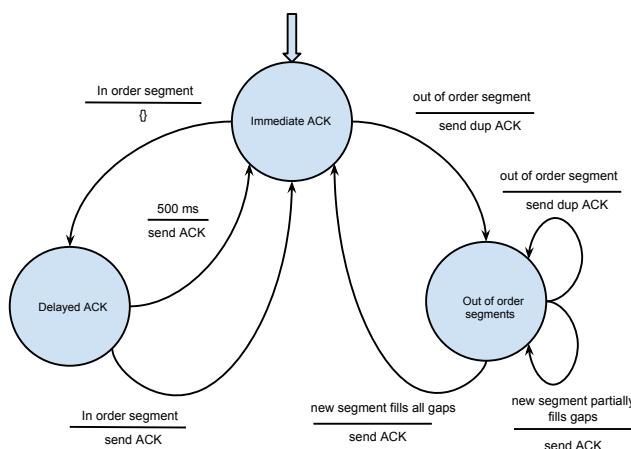


Figura 8.16: Funcionamento do receptor TCP

Na Listagem 8.1, apresentada acima, e nas listagens 8.3 e 8.4, apresentadas a seguir, aquando da actualização do valor de `congWnd` na sequência de um ACK, usa-se sempre o valor de MSS para actualizar aquela variável. Na realidade o valor usado é o número de bytes que o ACK cobre pois nem sempre o TCP envia segmentos da dimensão de MSS e, por outro lado, nem todos os segmentos recebidos desencadeiam o envio de um ACK pelo receptor. Para que o leitor os possa ter presentes, a Figura

8.16 recorda os aspectos essenciais do funcionamento do receptor TCP através da sua máquina de estados.

A seguir apresenta-se uma visão resumida das duas versões iniciais do algoritmo de controlo da saturação do TCP.

Controlo da saturação – TCP Tahoe

O primeiro algoritmo de controlo da saturação usado pelo protocolo TCP foi designado *TCP Tahoe congestion control* pois foi introduzido em 1988 com o sistema de operação Unix 4.2 BSD ou *release Tahoe*.

Inicialmente o TCP não conhece o valor de **sst** (*slow-start threshold*, o qual controla o ponto a partir do qual se passa da fase de incremento multiplicativo da janela para a fase de incremento aditivo) e usa um valor de **sst** alto (e.g., 64 KBytes) pelo que geralmente a janela sobe de forma multiplicativa até ao primeiro *timeout*. É como se o TCP começasse por tentar saber à partida qual o débito máximo possível ou, posto de forma irónica, para evitar a saturação é necessário começar por provocá-la. Na sequência do primeiro *timeout*, o valor de **sst** passa a 1/2 do valor de **congWnd** no momento do *timeout*, **congWnd** toma o valor inicial, i.e., **MSS**, e entra-se na fase *slow start* até **congWnd** atingir o valor de **sst**, altura em que se transita para a fase *congestion avoidance* (esta fase também se poderia ter chamado *network probing*) até disparar um novo *timeout* ou ter lugar o evento *triple duplicateACK* e o processo repete-se como está ilustrado na Figura 8.17. A Listagem 8.3 apresenta de forma mais completa a gestão dos valores de **congWnd** e de **sst**.

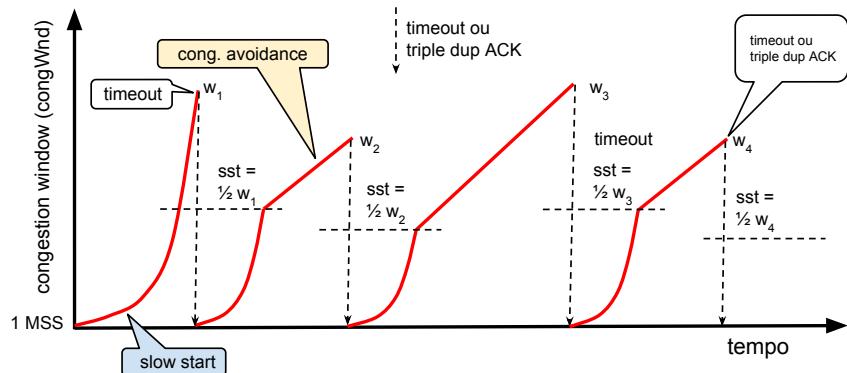


Figura 8.17: Evolução do valor do limite **congWnd** ao longo de uma conexão TCP na versão TCP Tahoe

Controlo de saturação – TCP Reno

Como vimos na secção 7.2, o TCP usa um método de cálculo do *timeout* que pretende prevenir, tanto quanto possível, as retransmissões prematuras, e é conservador na estimação do RTT e dos alarmes. Em particular, o valor do alarme duplica sempre que se retransmite um segmento. Isso quer dizer que em caso de perda de pacotes, muitos podem ter sido enviados antes da detecção da perda através do alarme. Por outro lado, para evitar tanto quanto possível retransmissões inúteis, o emissor usa também o mecanismo *FAST retransmission* quando recebe 3 ACKs duplicados (ou seja, depois de receber 4 ACKs seguidos com o mesmo número de sequência), o que é outra forma de detectar a perda de um segmento. Só que neste caso a perda foi detectada mais cedo e tudo indica que se trata de uma perda isolada, e não da perda

Listing 8.3: Pseudo código simplificado da actualização de `congWnd` na versão TCP Tahoe

```

On event < START >:
    congWnd = MSS
    sst = 64 KBytes
    dupAckCount = 0

On event < TIMEOUT or ( dup_ACK and dupAckCount == 3 ) >:
    sst = max (congWnd / 2, 2*MSS)
    congWnd = MSS
    dupAckCount = 0

On event < DUP_ACK and dupAckCount < 3 >:
    dupAckCount++

On event < NEW_ACK and status == slow start >:
    congWnd += MSS
    dupAckCount = 0

On event < NEW_ACK and status == congestion avoidance >:
    congWnd += MSS * MSS / congWnd
    dupAckCount = 0

```

de uma grande quantidade de segmentos, pois o emissor continua a receber ACKs o que mostra que os segmentos seguintes foram entregues ao receptor.

Por isso o algoritmo de controlo de saturação foi alterado e, quando é detectado um evento *triple duplicate ACK*, ao invés de se reduzir o valor de `congWnd` a um MSS, actualiza-se o valor de `sst` como anteriormente, mas entra-se num novo estado, chamado *fast recovery*, afecta-se a `congWnd` o valor `sst + 3 * MSS` e não o valor do MSS, e continua-se a manipular aditivamente o valor da `congWnd`.

Esta nova versão do algoritmo de controlo da saturação foi introduzida com a *release* do sistema de operação Unix 4.3 BSD Reno em 1990. Este algoritmo conduz à gestão de `congWnd` como ilustrado de forma aproximada na Figura 8.18 e descrito com mais detalhe na Listagem 8.4. Quando a conexão é estável e a grande maioria dos eventos de perda de segmentos são do tipo *triple duplicate ACK*, esta gestão conduz a um comportamento em “dente de serra”, compatível com a filosofia AIMD apresentada inicialmente.

No início do estado *fast recovery*, a variável `congWnd` toma o valor `congWnd = sst+3*MSS` porque continuaram a chegar dados ao receptor, e enquanto se permanecer nesse estado é porque se continuam a receber ACKs duplicados e a janela pode aumentar. Quando o processo de recuperação terminar, `congWnd` toma de novo o valor `congWnd = sst`. A máquina de estados com acções da Figura 8.19 mostra o comportamento do emissor TCP Reno.

O protocolo TCP recebeu diversos algoritmos de controlo da saturação com o objectivo de maximizar a capacidade da rede utilizada por cada conexão, sem deslealdade para as restantes com que compete, e evitando a entrada da rede em saturação. As versões iniciais desses algoritmos chamaram-se TCP Tahoe e TCP Reno e usam a perda de pacotes como sinónimo de *feedback* implícito da rede.

Estas versões constituem uma primeira solução para um problema complexo dada a diversidade das situações em que o controlo de saturação tem de actuar: quer a conexão atravesse uma pequena área da rede, homogénea, bem dimensionada, e com um pequeno RTT; quer a conexão atravesse uma grande quantidade de canais, hete-

Listing 8.4: Pseudo código simplificado da actualização de `congWnd` na versão TCP Reno

```

On event < START >:
    congWnd = MSS
    sst = 64 KBytes
    dupAckCount = 0
    status = slow start

On event < TIMEOUT >:
    sst = max (congWnd / 2, 2*MSS)
    congWnd = MSS
    dupAckCount = 0
    status = slow start

On event < NEW_ACK and status == slow start >:
    congWnd += MSS
    dupAckCount = 0
    if ( congWnd >= sst ) status = congestion avoidance

On event < NEW_ACK and status == congestion avoidance >:
    congWnd += MSS * MSS / congWnd
    dupAckCount = 0

On event < NEW_ACK and status == fast recovery >:
    congWnd = sst
    dupAckCount = 0
    status = congestion avoidance

On event < DUP_ACK and dupAckCount == 3 >: // triple dup ack
    sst = max (congWnd / 2, 2*MSS)
    congWnd = sst+3*MSS
    dupAckCount = 0
    status = fast recovery

On event < DUP_ACK and dupAckCount < 3 and status != fast recovery >:
    dupAckCount++

On event < DUP_ACK and status == fast recovery >:
    congWnd += MSS
    dupAckCount++

```

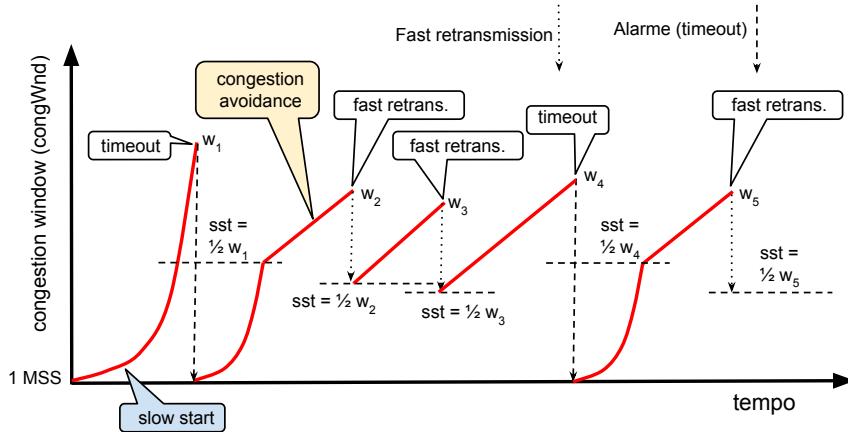


Figura 8.18: Evolução do valor do limite $cwnd$ ao longo de uma conexão TCP na versão TCP Reno (fase *fast recovery* omitida)

rogéneos, com níveis de utilização muito diversos e em presença de um elevado valor do RTT.

Acresce que, ao basearem-se exclusivamente em mecanismos de controlo com *feedback* implícito, os algoritmos de controlo da saturação requerem que a conexão TCP “tente adivinhar” o que se passa na rede a partir dos sintomas que pode inferir e não de avisos explícitos.

Os algoritmos de controlo da saturação usados com o protocolo TCP continuaram a evoluir desde a versão Reno, como a seguir veremos na secção 8.5, mas por agora vamos analisar outras soluções de suporte do controlo da saturação com base na colaboração da rede através da transmissão de notificações explícitas sobre o estado da mesma.

8.4 Controlo com *feedback* explícito

A saturação da rede é detectada pelos comutadores através da análise do estado das sua filas de espera. Quando a saturação se aproxima, os comutadores podem tentar avisar os originadores dos fluxos que os atravessam para que estes diminuam o ritmo de emissão de pacotes. Os inventores do protocolo IP imaginaram que seria possível, nessa situação, um comutador enviar uma notificação explícita para o emissor de cada pacote que fosse suprimido. Assim, quando um comutador suprimisse um pacote IP por não ter lugar para ele numa fila de espera, deveria enviar um pacote, chamado *choke packet*, para o endereço origem do pacote suprimido, e foi proposta a utilização do protocolo ICMP para esse efeito.

Se um comutador for atravessado por inúmeros fluxos de pacotes distintos, com diversas origens, em caso de congestionamento teria de enviar inúmeros *choke packets*. Se não o fizer, ou se estes sinais se perderem, a saturação persiste. Como é fácil de reconhecer, numa rede de grande escala, o envio de *choke packets* faz “parte do problema e não da solução”, na medida em que aumenta o tráfego numa rede já saturada e sobrecarrega comutadores já em apuros. Por outro lado, quando o mecanismo foi introduzido, não se sabia lá muito bem o que o receptor deveria fazer quando recebesse *choke packets*. Por isso este mecanismo nunca foi de facto implementado.

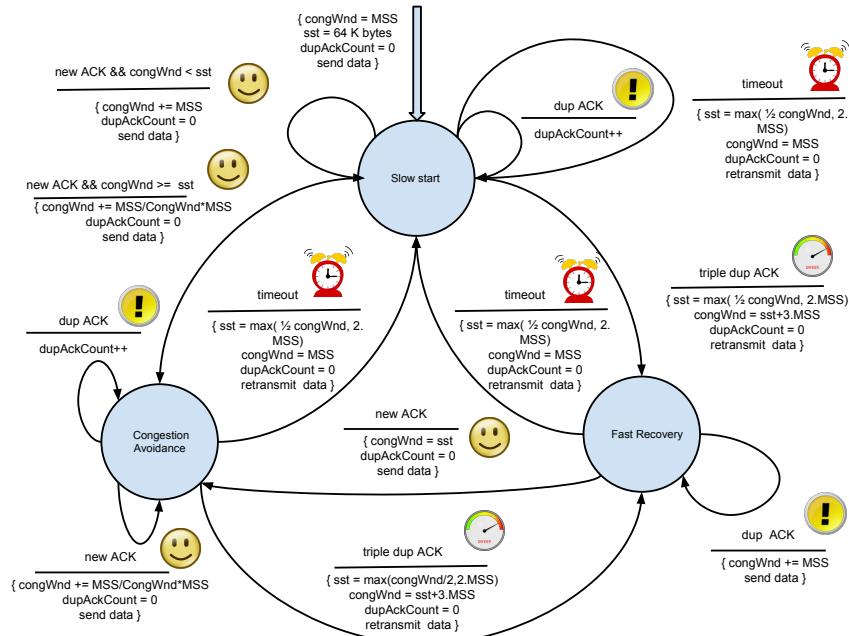


Figura 8.19: Descrição do algoritmo de controlo da saturação TCP Reno

No entanto, como a maioria dos fluxos (ou dos circuitos) estão associados a tráfego nos dois sentidos como o TCP, o *feedback* da rede pode chegar ao emissor original encavalitado no tráfego enviado no sentido contrário pelo receptor. Se possível de aplicar, esta técnica resolve o problema mesmo quando os caminhos num e outro sentido são diferentes e não implica a introdução de pacotes extra na rede.

A Figura 8.20 ilustra a ideia. Se um comutador quiser transmitir alguma informação sobre o estado da rede ao emissor dos pacotes, anota essa informação nos cabeçalhos dos pacotes que este enviou, e o receptor, quando enviar pacotes no sentido contrário, copia a informação da rede e coloca-a no cabeçalho dos pacotes que envia ao emissor original. Por exemplo, se um comutador quiser enviar informação a um emissor TCP, anota essa informação no cabeçalho dos segmentos que encaminha, e o receptor ecoa-a para o emissor original no cabeçalho dos segmentos TCP que envia no sentido contrário. Naturalmente, esta técnica só pode ser usada com tráfego bidireccional e com a colaboração da outra extremidade da conexão.

Este mecanismo foi introduzido inicialmente em redes de circuitos e usava pacotes de controlo especiais para transportar a informação de controlo de volta ao emissor original. Quando recebiam os pacotes de controlo, os comutadores podiam analisar o seu conteúdo, eventualmente actualizá-lo e, quando o pacote de controlo chegasse à extremidade do circuito era ecoado para a extremidade contrária, levando a informação recolhida no sentido inverso para a extremidade a quem a mesma interessava.

ECN - Notificação explícita da saturação

Inspirado nesta ideia foi introduzido no protocolo TCP um mecanismo de sinalização explícita, da existência de saturação, ao emissor de um fluxo TCP. O mecanismo chama-se ECN - *Explicit Congestion Notification*, ver o RFC 3168, e utiliza duas flags do cabeçalho IP e outras duas do cabeçalho TCP. A implementação pressupõe a colaboração de diversas componentes da rede. Os comutadores usam as flags do

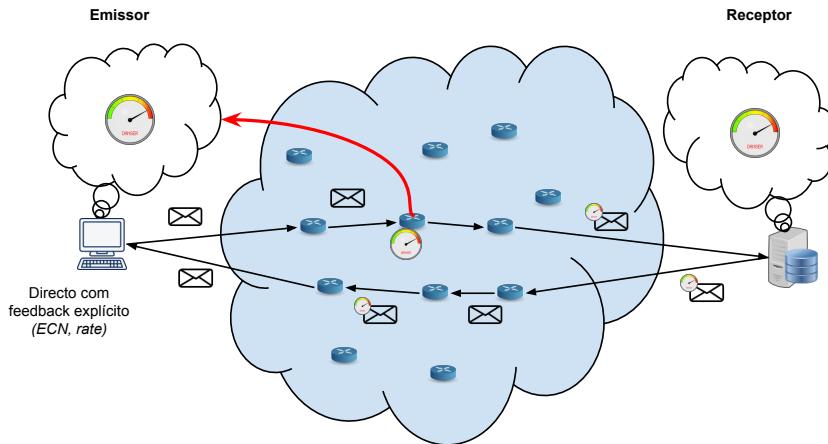


Figura 8.20: Encaminhamento de *feedback* explícito encavalitado no tráfego enviado no sentido contrário

cabeçalho IP para, no caso de a saturação se aproximar, marcarem um pacote com essa informação. Quando o pacote chega ao destino e o nível TCP envia um segmento no sentido contrário, usa *flags* do cabeçalho TCP para indicar ao emissor que um segmento que recebeu passou por um comutador em *stress*. Trata-se portanto de um mecanismo que envolve alguma colaboração explícita entre o nível rede e o nível transporte.

Assim, na abertura da conexão, ambas as extremidades indicam se suportam ou não ECN, pois não vale a pena perder tempo com o mecanismo se a outra não o suportar. Caso ambas o suportem, o emissor quando envia segmentos para a outra extremidade, indica no cabeçalho do pacote IP, no qual o segmento é encapsulado, que esse pacote transporta dados de uma conexão de transporte que suporta ECN.

No seu percurso para o receptor, o pacote com o segmento TCP passa por diversos comutadores. Se algum deles estiver em vias de ficar saturado e suportar a opção ECN (nem todos os comutadores a suportam), marca esse facto no cabeçalho do pacote IP. Se mais do que um comutador estiver nessa situação, basta que o primeiro marque. Quando o pacote chega ao receptor e é entregue ao nível TCP, como este suporta ECN, verifica se o pacote que contém o segmento que acabou de receber tem alguma indicação sobre saturação e, se for o caso, quando enviar tráfego no sentido contrário, por exemplo ao enviar um ACK, assinala no cabeçalho TCP desse ACK que o segmento com dados recebido continha uma notificação de saturação.

Quando este sinal chegar ao emissor original, este realiza exactamente as mesmas operações, do ponto de vista do controlo da saturação, que as que executa quando recebe um triplo ACK duplicado (mas sem retransmitir os dados pois estes foram recebidos).

A Figura 8.21 ilustra o mecanismo. Quando o emissor TCP envia dados novos⁷ marca os pacotes IP com esses segmentos com a indicação de que gostaria de ser notificado, se possível, através do mecanismo ECN (legenda (1) na figura). Caso o pacote chegue ao receptor com a indicação de que um comutador intermédio está saturado (legendas (2) e (3) na figura), o receptor envia segmentos TCP para o emissor (com os ACKs por exemplo) com a *flag* TCP ECE (ECE - ECN Echo) posicionada, a

⁷Com retransmissões ou ACKs o mecanismo não é usado.

indicar que foi avisado de que há risco de saturação no sentido contrário (legenda (4) na figura). O emissor, depois de receber a notificação ECN, nos novos segmentos que enviar ao receptor, posiciona a *flag* TCP CWR (CWR - *Congestion Window Reduced*), que funciona como uma espécie de confirmação de recepção da notificação, ou ACK, de saturação para o receptor, que a deixará de enviar nos segmentos seguintes.

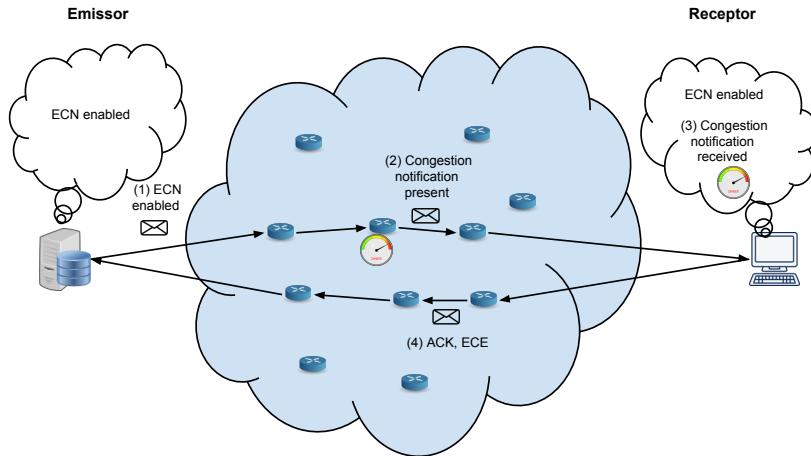


Figura 8.21: Funcionamento do mecanismo TCP ECN

O mecanismo é um exemplo de *feedback* explícito da rede aos computadores sobre a sua situação do ponto de vista da saturação. Este mecanismo é binário, mas existem outros mecanismos, não presentes nas redes IP, em que esse *feedback* é mais completo, e contém recomendações com indicação de qual o débito a usar pelo emissor.

Um mecanismo desse tipo poderia ser introduzido utilizando o campo de opções do TCP. O problema é que essa solução teria dois problemas delicados de implementação. O primeiro seria a obrigatoriedade de os comutadores conhecerem os detalhes do protocolo TCP, o que seria uma violação grosseira da *separation of concerns* implementada pela separação da rede em níveis. De facto, com essa opção, qualquer alteração de um protocolo de transporte poderia ter impacto ao nível rede. O outro problema, também delicado, é o de determinar qual o débito a oferecer a cada um dos fluxos. Isso só é fácil de determinar quando os fluxos são poucos e todos da mesma natureza, ou quando se reserva capacidade *a priori* como é possível nas redes de circuitos.

O mecanismo ECN Notificação Explícita da Saturação (ECN – *Explicit Congestion Notification*) permite indicar explicitamente aos emissores que devem refrear o seu débito de emissão para não saturarem a rede. Infelizmente, requer a cooperação entre os comutadores e o nível transporte nos computadores e a sua adopção tem-se revelado lenta. Trata-se de uma implementação pragmática do *feedback* explícito, mais simples de implementar do que um mecanismo que indicasse créditos concretos.

Vamos a seguir ver um mecanismo de *feedback* implícito chamado RED, uma alternativa a ECN.

Supressão prematura de pacotes

O mecanismo ECN é interessante mas exige que os comutadores e os protocolos de transporte o suportem. Quase duas dezenas de anos depois de o mecanismo ter sido proposto para as redes TCP/IP, estima-se que apenas uma minoria dos comutadores IP o suportam, ver por exemplo a análise apresentada em [Grosvenor et al., 2015]. O mais fácil é mesmo suprimir pacotes quando há saturação (na verdade nessa altura não existem muitas outras alternativas!).

No entanto, não será melhor começar a suprimir pacotes antes de a situação ser dramática? Se quando a carga das filas de espera de um comutador começar a passar para lá do razoável, este suprimir prematuramente alguns pacotes, tomados ao acaso e distribuídos por diferentes fluxos, isso funcionaria como um aviso prévio para as extremidades dos fluxos que o melhor era começarem já a reduzir o débito, pois a carga da rede parece estar a aumentar para lá do razoável. Como quando perde um segmento isolado, o TCP executa o procedimento “*fast recovery*” (do controlo de saturação Reno e seus sucessores) e abranda o ritmo de emissão, e dado que o TCP é um dos protocolos mais usados, essa estratégia pode contribuir para diminuir a carga (e a potencial saturação) de toda a rede.

A mesma linha de raciocínio recomenda que a probabilidade de um pacote ser suprimido seja proporcional ao nível de saturação: muito elevada se esse nível for elevado, pequena, ou mesmo nula, no caso contrário. Por outro lado, é igualmente importante distribuir as supressões prematuras de pacotes pelos diferentes fluxos, com maior incidência nos de maior débito.

Esta ideia foi proposta por Sally Floyd e Van Jacobson [Floyd and Jacobson, 1993], foi baptizada RED (*Random Early Detection* ou *Random Early Drop*) e é implementada da seguinte forma. Quando um pacote chega a um comutador, este determina o estado da fila de espera em que o pacote vai ser colocado à espera de ser transmitido. Nessa altura é tomada a decisão de supressão ou aceitação do pacote usando um gerador de números pseudo-aleatórios. Suprimir pacotes aleatoriamente aumenta a probabilidade de distribuir os avisos pelos diferentes fluxos, com maior incidência naqueles que enviam mais pacotes, ou seja nos com maior débito.

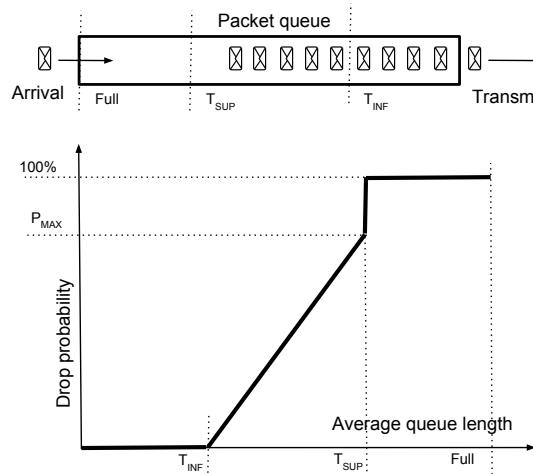


Figura 8.22: Função probabilidade de supressão de pacotes com RED

A supressão ou não de um pacote depende da dimensão da fila de espera e da função probabilidade que é usada para tomar a decisão. A dimensão da fila de espera usada

pelo RED não é a dimensão exacta, mas uma média calculada através de uma EWMA (Exponential Weighted Moving Average) semelhante à usada pelo TCP para calcular o RTT, ver a Secção 7.2. O objectivo é evitar que numa situação em que o tráfego médio é baixo, um pico momentâneo, que pode ser absorvido, seja excessivamente penalizado.

A função probabilidade do algoritmo RED usa diversos parâmetros, ver a Figura 8.22, nomeadamente a ocupação média da fila de espera abaixo da qual aos pacotes não devem ser suprimidos ($Threshold_{INF} = T_{INF}$), a ocupação média acima da qual todos os pacotes devem ser sempre suprimidos (T_{SUP}), e o andamento intermédio dessa probabilidade (dependente do valor de P_{max}) que determina a inclinação da recta intermédia. Se o valor de P_{MAX} for baixo, a probabilidade de suprimir pacotes quando a ocupação é média, i.e., entre T_{INF} e T_{SUP} nunca é muito alta. Quando $P_{MAX} = 1$, a distribuição das supressões é uniforme entre T_{INF} e T_{SUP} . A análise experimental permitiu concluir também que a probabilidade de suprimir um pacote deveria ir aumentando conforme aumentasse o número de pacotes aceites desde a última supressão, ou seja, a probabilidade deveria aumentar conforme foi passando mais tempo sem haver nenhuma supressão prematura. A listagem 8.5 apresenta de forma mais detalhada a forma de cálculo dos diferentes parâmetros.

Listing 8.5: Pseudo código da actualização dos parâmetros do algoritmo RED na sequência da recepção pelo comutador de mais um pacote

```

thresholdDiff = thresholdMax - thresholdInf
count = 0 // number of sent packets in sequence

// "gamma" defines the weight of the historic average when calculating
// the new average, i.e., it adjusts the sensibility to sudden changes
averageQueue = (1 - gamma) * averageQueue + gamma * currentMeasuredQueue

if (averageQueue < thresholdInf) dropProbability = 0
else if (averageQueue > thresholdMax) dropProbability = 1
else {
    p = pMax * (averageQueue - thresholdInf) / thresholdDiff
    dropProbability = p / (1 - count * p)
}

random = getRandom() // between 0 and 1
if (random < dropProbability) {
    drop packet
    count = 0
} else { count++ }

```

RED é uma técnica necessariamente menos interessante do que ECN porque suprime pacotes, mas é mais geral e pode ser usada em qualquer comutador e com qualquer transporte. O problema principal consiste em afinar os seus parâmetros, em função da quantidade de fluxos, da sua heterogeneidade, etc. de tal forma que no fim do dia se ganhe mais em evitar a saturação por supressão prematura de pacotes, do que sem usar o mecanismo. O estudo desses parâmetros é muito difícil sobre um protótipo onde se procure recriar as condições reais que ocorrem na Internet. A sua realização por simulação também é bastante difícil [Floyd and Paxson, 2001].

Em alternativa à notificação explícita da saturação, é possível suprimir prematuramente pacotes, na esperança que os respectivos emissores reduzam de forma equitativa o seu débito, como almeja o mecanismo de **Supressão prematura aleatória de pacotes (RED – Random Early Detection)**. Infelizmente, este mecanismo, simples de concretizar, é difícil de parametrizar de forma a que o mesmo se adapte dinamicamente à situação da rede e da diversidade dos fluxos e tenha mais proveito do que custos.

8.5 Equidade e desempenho do TCP

Caracterizar o desempenho do protocolo TCP tendo em consideração o seu algoritmo de controlo da saturação envolve várias facetas. Por um lado, é necessário analisar se a estratégia seguida pelo algoritmo conduz a uma distribuição equitativa da capacidade da rede entre os diferentes fluxos que a atravessam. Por outro lado é necessário analisar também o impacto do algoritmo sobre o desempenho de um fluxo tendo em consideração a capacidade disponível para o mesmo no *bottleneck link*. Começaremos por discutir o primeiro aspecto.

Equidade

Como vimos na secção 8.2, no contexto do controlo de saturação do TCP só faz sentido discutir equidade face à divisão da capacidade do *bottleneck link*. A pergunta que se coloca então é saber se perante várias conexões TCP que competem pela capacidade do mesmo *bottleneck link*, a estratégia AIMD (*Additive Increase Multiplicative Decrease*) conduz ou não a uma divisão equitativa dessa capacidade.

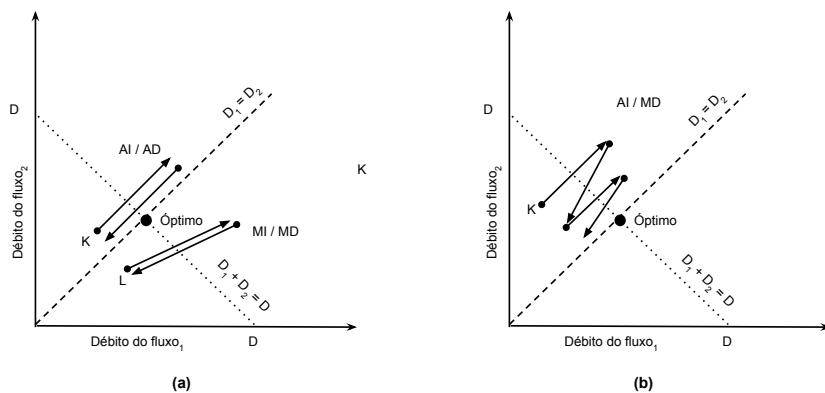


Figura 8.23: O mecanismo AIMD faz com que dois fluxos tendam para o equilíbrio

A resposta à questão no caso geral é difícil, mas no caso particular em que duas conexões TCP, com o mesmo RTT, partilham um *bottleneck link* com o débito D , a resposta é afirmativa. Chiu e Jain [Chiu and Jain, 1989] desenvolveram um esquema particularmente elegante para o mostrar, que se apresenta na Figura 8.23. Na discussão que se segue admite-se igualmente que ambas as conexões detectam os eventos *timeout* ou *triple duplicate ACK* simultaneamente.

Comecemos por perceber o gráfico (a), à esquerda. Qualquer ponto sobre esse gráfico representa um caso particular de partilha do débito D entre as duas conexões. A linha a ponteado, com a etiqueta $D_1 + D_2 = D$, representa as alternativas de partilha que consomem todo o débito do canal. A linha oblíqua a tracejado, com a etiqueta $D_1 = D_2$, representa os pontos em que a divisão do débito do canal é equitativa. O ponto de interceção das duas linhas corresponde à afectação óptima pois esta é equitativa e toda a capacidade do canal é afectada. Os pontos abaixo da linha a ponteado são afectações possíveis da capacidade pois representam pontos tais

que $D_1 + D_2 \leq D$, os pontos acima dessa linha conduzem a saturação visto que são afectações impossíveis pois $D_1 + D_2 > D$.

Dado o ponto K , que representa uma afectação não óptima visto que não é equitativa nem consome toda a capacidade disponível, a estratégia incremento e decremento aditivos (AIAD) aplicada a esse ponto, levaria as suas coordenadas (correspondentes aos débitos afectados aos fluxos 1 e 2) a serem ambas incrementadas ou decrementadas do mesmo valor, o que levaria K a dar lugar a um novo ponto em que ambas as coordenadas tinham sido incrementadas ou decrementadas do mesmo valor. Logo, esse novo ponto estará necessariamente sobre uma linha paralela à linha a tracejado e não se aproxima desta. Assim, o incremento ou o decremento aditivos não aproxima a solução da equidade, apenas afecta ou retira a mesma capacidade a cada uma das conexões e a ausência de equidade da afectação mantém-se.

Em contrapartida, o incremento e recuo multiplicativos (MIMD), como por exemplo o aplicado ao ponto L , mantém a proporção entre as duas coordenadas, pelo que a sua aplicação faz deslocar L ao longo de um segmento de recta com origem no ponto $(0, 0)$, correspondente à afectação (nulo, nulo) a cada uma das conexões. O incremento e o decrecimento de capacidade de forma multiplicativa só aproxima a afectação da equidade quando as capacidades afectadas decrescem, aproximando-se de 0, o que não é lá muito interessante. Assim, a estratégia AIAD (incremento e decremento aditivos) não aproxima a solução da equidade. A estratégia MIMD (incremento e decrecimento multiplicativos) só aproxima a partilha da equidade quando a afectação total se aproxima de 0. É a equidade do cemitério, onde todos estão igualmente mortos.

Se atentarmos agora no gráfico (b), à direita, partindo do ponto K aplica-se a estratégia AIMD, verifica-se que sempre que se sobe apenas se afecta mais capacidade a ambas as conexões sem corrigir a equidade, mas quando se desce o ponto aproxima-se da semi recta a tracejado dos pontos caracterizados por $D_1 = D_2$. Assim, no limite, a aplicação sucessiva desta estratégia coloca o ponto a oscilar sobre a linha a tracejado em que a afectação é equitativa e converge para o ponto da afectação óptima. A estratégia MIAD conduz ao afastamento da linha a tracejado e é portanto não equitativa.

A argumentação desenvolvida pelos autores Chin e Jain põe em evidência que, ao contrário das outras, a estratégia AIMD conduz à estabilidade e à equidade. De forma intuitiva também é fácil reconhecer que se trata de uma estratégia que trata a rede com prudência (AI) e que foge rapidamente da saturação (MD).

O argumento desenvolvido aplica-se a mais do que duas conexões mas deixa de poder aplicar-se sempre que o RTT das conexões é diferente ou as conexões TCP entram em competição com outros fluxos de pacotes que não aplicam de todo estratégias de controlo da saturação. A velocidade com que uma conexão TCP aumenta o seu débito durante a fase AI é tanto mais rápida quanto menor for o seu RTT, portanto as conexões com um menor RTT consomem uma fração superior da capacidade do canal quando estão em competição com outras com um RTT mais elevado. Mais: dado que estas serão mais frequentemente responsáveis pela saturação do canal, empurram as outras para baixo, visto que ambas recuam proporcionalmente ao ponto onde estavam, e quando recomeçarem a subir, as de menor RTT sobem de novo mais depressa do que as outras.

Por outro lado, se as conexões TCP entrarem em competição pela capacidade de um canal com fluxos de pacotes de débito fixo, que não aplicam nenhuma estratégia adaptativa, necessariamente perdem “a guerra”, pois os fluxos TCP diminuem o seu débito, enquanto os outros fluxos não. Por exemplo, se vários fluxos TCP competirem por um canal de débito D com fluxos UDP que consumem $1/2D$, os fluxos UDP consumirão sempre o débito que usam ($1/2D$) enquanto que os fluxos TCP apenas competirão pela capacidade que o UDP deixa livre. No limite, se os fluxos UDP consumirem toda a capacidade D , as conexões TCP praticamente “morrem”, pois só conseguirão tentar enviar um único segmento por RTT, e como este frequentemente se

perde, só tentam de novo após disparar o *timeout*, mas como de cada vez duplicam o valor do temporizador usado, a situação só poderá agravar-se até que só tentam enviar (melhor dizendo reenviar) um segmento de vários em vários segundos.

Finalmente, para terminarmos esta discussão sobre a equidade, é necessário ter em atenção que a divisão do débito do *bottleneck link* é feita por conexão TCP e não por computador. Assim, uma maneira de obter uma fração superior do débito partilhado é abrir várias conexões em paralelo ao invés de uma só. Esta estratégia é usada com frequência por aplicações P2P de partilha de conteúdos.

É possível demonstrar que a estratégia AIMD tende para a estabilidade e equidade na partilha de um *bottleneck link* entre várias conexões TCP com o mesmo RTT. No entanto, se as conexões TCP tiverem RTTs distintos a equidade não é garantida.

Por outro lado, o TCP não pode competir com fluxos de pacotes que não aplicuem a mesma estratégia de controlo de saturação, pois reduz sempre o débito de emissão independentemente da estratégia seguida pelos seus competidores.

Isto mostra igualmente que quando se introduzem alterações no protocolo TCP, é necessário garantir que estas mantêm a equidade para com as versões anteriores do protocolo.

Após termos olhado de perto a problemática da equidade, vamos a seguir discutir o problema do desempenho efectivo de uma conexão TCP que está a usar o algoritmo de controlo da saturação Reno.

Desempenho

No final da secção 7.3 o débito médio extremo a extremo de uma conexão TCP foi grosseiramente caracterizado como sendo proporcional a W/RTT , em que W representa a dimensão média da janela em bits e RTT tem o significado habitual. O problema é que, como sabemos agora, W varia todo o tempo. Por outro lado, varia de forma diferente conforme a duração da conexão e o comportamento dos competidores.

Para reanalisarmos a questão vamos usar de novo um modelo relativamente simples. Admitamos que a conexão tem uma grande duração e que está continuamente a transferir dados. Admitamos também que o *bottleneck link* está no interior da rede. Com efeito, se o *bottleneck link* for o canal que liga o emissor à rede, a velocidade de transferência extrema a extremo será a desse canal, pois os mecanismos de gestão dos *buffers* do computador emissor leváram a que o ritmo de emissão seja o permitido por esse canal e, nessa situação, em princípio, não ocorre perda de pacotes por saturação. Admitamos igualmente que a dimensão da janela do receptor não limita a dimensão da janela do emissor através de controlo de fluxos. Podemos também considerar que a maioria das perdas de pacotes são tratadas por *fast retransmit* e desprezarmos a ocorrência de *timeouts*.

Com estas hipóteses, o débito de extremo a extremo medido em termos do quociente W/RTT terá um andamento semelhante ao sugerido pelo gráfico da Figura 8.24 e estará compreendido no intervalo $1/2D$ e D , em que D representa a fração do débito do *bottleneck link* que cabe à conexão. Uma estimativa grosseira indica que em média, estimada a longo prazo, o débito médio extremo a extremo deverá ser $\approx 3/4D$.

O modelo usado é muito simples e usa hipóteses muito fortes: D é constante e os competidores são igualmente estáveis, a conexão é muito longa e tem sempre dados

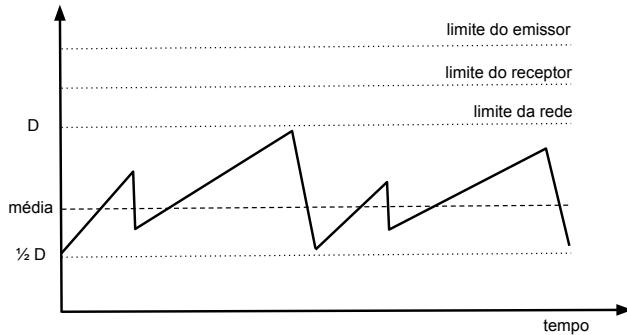


Figura 8.24: Débito de uma conexão TCP idealizada

para transmitir, não surgem novas conexões (que durante a fase *slow start* inicial provocam necessariamente saturação), não há perdas de pacotes a não ser esporádicas, a aplicação e o receptor não impõem limitações mais estritas do que a rede ao débito, e a capacidade do canal que liga o emissor à rede não refreia o seu débito de emissão. Naturalmente, o modelo não se aplica quando essas hipóteses não são verdadeiras.

É interessante discutir várias situações em que o modelo acima não se aplica pois algumas são frequentes e relevantes.

Conexões muito curtas

O acesso à Web é feito sobre TCP e frequentemente são abertas conexões para servidores para obter objectos de pequena dimensão (20 KBytes por exemplo). Logo que o cliente obtém o objecto fecha a conexão. Com um MSS de 1250 bytes, 20 KBytes correspondem a 16 segmentos. Devido à abertura de conexão e ao mecanismo *slow start*, no fim do 2º RTT, o cliente enviou o pedido e recebeu 1 segmento, no fim do 3º, recebeu 3 segmentos, no fim do 4º, recebeu 7 segmentos, no fim do 5º, recebeu 15 segmentos e no fim do 6º, já recebeu a totalidade dos 16 segmentos.

Em muitas situações a janela inicial poderia ser superior a 1 segmento. Por exemplo, se a janela inicial começasse com 4 segmentos, no fim do 2º RTT, o cliente já teria recebido 4 segmentos, no fim do 3º, já teria recebido 12 segmentos e no fim do 4º já teria recebido até 28 segmentos. O tempo para obtenção do objecto baixaria para 66% do anterior apenas porque a janela começou com 4 segmentos. Se o RTT for 100 ms, o que pode acontecer no acesso a servidores remotos, o objecto seria obtido em 400 ms ao invés de em 600 ms. A conexão TCP normal colocou a hipótese inicial de dispor apenas da capacidade de $1250 \times 8/0,1 \approx 100$ Kbps no *bottleneck link*, tendo chegado depois a usar 800 Kbps, enquanto que no segundo caso colocaria a hipótese de dispor inicialmente de 400 Kbps e depois de 800 Kbps.

Sempre que for possível começar com janelas maiores, tal é muito interessante para conexões de pequena duração. Na Internet actual a disponibilidade das capacidades indicadas é vulgar. No entanto, o que aconteceria em situações em que o *bottleneck link* não tivesse essas capacidades disponíveis? A simples abertura da conexão provocaria saturação. Idealmente deveria ser possível durante a abertura da conexão ter uma melhor estimativa da capacidade disponível para a mesma. De qualquer forma, as versões actuais de TCP usam janelas iniciais com a dimensão de vários segmentos.

O RFC 3390 recomenda que a janela inicial seja assim dimensionada

$$\min(4 \times MSS, \max(2 \times MSS, 4380 \text{ bytes}))$$

o que conduz com frequência a janelas de 4380 bytes, que correspondem geralmente a 3 segmentos pois 1460 bytes é um valor comum para o MSS. No entanto, o mesmo RFC abre a hipótese de se usarem janelas ainda maiores, o que se tornou prática corrente sobretudo quando se detectam RTTs elevados durante a abertura da conexão.

Perdas consecutivas

O algoritmo Reno baseia-se na hipótese de que quando a saturação se aproxima, têm lugar algumas perdas esporádicas de pacotes detectadas pelo evento *triple duplicate ACK*. No entanto, verificou-se que existem situações em que este tipo de perdas são consecutivas.

Nesta situação o algoritmo *fast recovery* do Reno tem como repercussão a descida exponencial da janela de saturação. Com efeito, com perdas sucessivas, o valor da variável `sst` diminui para próximo de alguns MSSs, o valor do *timeout* aumenta muito, e a conexão, mesmo se deixasse de perder pacotes a seguir, permaneceria muito tempo no estado *congestion avoidance* em que a janela é incrementada de forma linear.

Para lidar com este problema foram introduzidos dois algoritmos novos, um integrado com o tratamento dos ACKs selectivos chamado TCP SACK, e outro chamado New Reno.

Canais sem fios

O algoritmo de controlo da saturação põe a hipótese de que a maioria das perdas de pacotes deve a saturação e não a erros. Esta hipótese verifica-se nos canais com fios e sobretudo nos canais baseados em fibra óptica. No entanto, os canais sem fios têm uma elevada taxa de erros, o que frequentemente implica perda de pacotes por erros nos canais.

O TCP não consegue distinguir os diferentes tipos de perda de pacotes e interpreta-os a todos da mesma forma. Muitas vezes isso implica o disparo de um alarme e a redução da dimensão da janela a um segmento, ou pelo menos a sua redução a metade se o erro for mais esporádico.

É difícil encontrar uma versão do algoritmo que lide bem com este tipo de situações. Acresce que a prática mostrou que as redes celulares sem fios introduzem RTTs artificialmente elevados, o que prejudica a recuperação do TCP numa situação em que a janela foi inutilmente reduzida. Para uma discussão preliminar destes problemas em redes sem fios o leitor poderá consultar [Mascolo et al., 2001], onde os autores propõem um método de estimar a capacidade efectivamente disponível para a conexão TCP o que permite controlar de forma mais rigorosa e suave as variações da janela do emissor.

Soluções nesta linha voltarão a ser discutidas no fim da secção onde são apresentados algoritmos de estimativa da capacidade extremo a extremo realmente disponível. Essa estimativa, se rigorosa, permite evitar a congestão e o recuo excessivo da dimensão da janela quando há perdas esporádicas de pacotes.

Não se têm encontrado soluções totalmente satisfatórias usando controlo da saturação tradicional, baseado em *feedback* implícito, para situações em que ocorrem erros esporádicos. As alternativas de maior sucesso e independentes do TCP consistem em tentar mascarar os erros dos canais sem fios através de mecanismos de recuperação dos erros directamente na extremidade desses canais, e portanto transparentes para os níveis superiores. Infelizmente, essas soluções aumentam ainda mais o tempo de trânsito. A outra alternativa é colocar *caches* dentro das redes móveis celulares para que seja desnecessário, se possível, ter conexões com elevado RTT e atravessando muitos canais distintos. Outra alternativa consiste em “partir” a conexão em duas, uma correspondente ao canal sem fios, e outra correspondente aos restantes canais.

Trata-se de mais um caso que ilustra que os *end-to-end arguments* não se podem aplicar de forma simplista.

Reordenação de pacotes

Apesar de a qualidade de serviço do tipo “melhor esforço”, que caracteriza o protocolo IP, comportar a reordenação de pacotes, os algoritmos Tahoe e Reno interpretam a reordenação como uma perda de pacotes que desencadeia eventualmente um evento *triple duplicate ACK*. Existem situações de alteração do encaminhamento, que podem conduzir a reordenação de pacotes que são de facto esporádicas. No entanto, podem também suceder situações de distribuição de carga no interior da rede que provoquem reordenação de pacotes. O comportamento do TCP nessas situações conduz a uma redução dramática do desempenho da conexão.

Foram feitas diversas alterações dos algoritmos para lidar com este problema e também se tomaram medidas para que a distribuição de carga no interior da rede fosse feita de tal forma que diminuísse drasticamente a probabilidade de ocorrer reordenação de pacotes.

Ausência de equidade com diferentes RTTs – *RTT unfairness*

Diversos algoritmos atacam este problema e incluem alterações para tentar minorar os seus efeitos. A estratégia comum a esses algoritmos consiste em acelerar a subida da janela de saturação durante a fase *congestion avoidance*, sem no entanto introduzir uma subida multiplicativa. Desta forma as conexões com RTTs grandes são menos penalizadas face a conexões com RTTs curtos, as quais sobem mais rapidamente a sua janela de saturação durante a fase *congestion avoidance*.

Esta estratégia é também aplicada para atacar o problema a seguir apresentado.

Canais com grande volume

O TCP é usado para transferir dados para *smartphones* ligados por canais com baixa capacidade, mas também para transferir objectos de grande dimensão entre centros de dados. As redes que interligam esses centros de dados dispõem de canais dedicados com débitos muito elevados (*e.g.*, 10 Gbps) e um tempo de trânsito de centenas de milissegundos pois os centros de dados estão em diferentes continentes. Como se porta o TCP nesses canais?

Com um MSS de 1250 bytes (10^4 bits) e um RTT de 100 ms, para encher um canal com a capacidade de 10 Gbps (10^{10} bits por segundo), ou seja para encher o seu volume, são necessárias janelas com 10^9 bits, logo com 100.000 segmentos. Se a dimensão da janela a certa altura for metade, 50.000 segmentos, o algoritmo de incremento aditivo junta à janela mais 10 segmentos em cada segundo (1 segmento por RTT), logo seriam necessários 5.000 segundos sem erros (uma hora tem 3.600 segundos) para que a janela subisse até próximo de uma dimensão que permitisse explorar a capacidade disponível. Claramente, a parte que lida com o controlo da saturação do TCP Reno não chega para lidar com canais de grande capacidade e um elevado RTT.

Existe um resultado teórico que indica que a velocidade média de transferência extremo a extremo do TCP (débito extremo a extremo ou $D_{end-to-end}$) pode ser estimado pela equação

$$D_{end-to-end} = \frac{1.22 \times MSS}{RTT \times \sqrt{\rho}}$$

onde ρ representa a taxa de pacotes recebidos com erros. Se o RTT for da ordem de grandeza de 100 ms, o MSS da ordem de grandeza de 12.000 bits, para se atingirem velocidades de transmissão da ordem de grandeza de 10 Gbps é necessário que haja, no máximo, um erro em cada 5.000.000.000 segmentos o que é uma taxa praticamente impossível mesmo com fibra óptica.

Mesmo que os canais tenham capacidades bastante inferiores disponíveis para a conexão TCP, torna-se claro que os algoritmos de controlo da saturação com base em *feedback* implícito e na perda de segmentos devem ser melhorados em situações onde são necessárias janelas de grande dimensão para obter um bom rendimento. Por essa

razão existem inúmeros trabalhos para melhorar o desempenho do TCP em geral e nestas situações em particular.

Evitar a saturação

Num quadro em que a rede continue a não dar *feedback* explícito sobre o débito recomendado para uma conexão (o que evitaria que a conexão tenha que provocar saturação para o estimar), a alternativa consiste em tentar estimar esse débito usando o RTT e o débito real, assim como as respectivas variações. Adicionalmente, isso resolveria o problema do funcionamento do TCP sobre canais com grande volume. Dual e Vegas são dois algoritmos baseados nesta ideia.

O TCP Dual estima o RTT mínimo e máximo da conexão. O mínimo é o RTT mais curto observado, por exemplo na abertura da conexão, e o máximo é o RTT observado durante os períodos de saturação. Naturalmente, a diferença entre os dois dá uma indicação sobre o tempo perdido pelos pacotes em filas de espera. Finalmente, um determinado factor α , entre 0 e 1, é usado para estimar um valor limite da diferença entre o RTT observado e o RTT mínimo a partir do qual a saturação está próxima. Os autores do TCP Dual usavam $\alpha = 1/2$.

Quando a diferença entre o RTT corrente e o RTT mínimo ultrapassa o limite, a janela de saturação é descida de $1/8$. Senão, a janela pode crescer aditivamente. O algoritmo melhora claramente o desempenho de uma conexão única, mas revelou-se incapaz de assegurar equidade pois as conexões são incapazes de estimar correctamente o RTT mínimo correspondente a uma rede sem carga.

Com efeito, a primeira conexão que encontrasse a rede sem carga estimaria correctamente o RTT mínimo, mas uma conexão que tivesse início com a rede carregada, não conseguiria estimar tal RTT. Por outro lado, o método de estimar o melhor desempenho só a partir do RTT revelou-se insuficiente. O TCP Vegas tentou melhorar diversos desses aspectos.

Tal como o TCP Dual, o TCP Vegas tenta igualmente evitar a saturação (*congestion avoidance*) estimando um valor de janela de saturação correspondente ao valor máximo que não sature a rede.

Com este objectivo, Vegas estima periodicamente o valor do RTT base, que é um valor de RTT mínimo mas que é ajustado periodicamente. A partir desse RTT base, o valor de `congWnd` usado em cada momento permite calcular

$$\text{expectedRate} = \text{congWnd}/\text{RTT}_{\text{base}}$$

que corresponde ao débito potencial quando a janela tem o valor `congWnd`, ver a Figura 8.25.

No entanto, contando o número de bytes transmitidos desde que um segmento é emitido até que o seu ACK chegue, numa situação sem perda de pacotes, é possível calcular o `actualRate` e o seu valor estabiliza a partir do momento em que a conexão está a usar a sua fração do *bottleneck link*, o que permite calcular a janela correspondente a esse débito.

A seguir a diferença $diff = \text{expectedRate} - \text{actualRate}$ é calculada. O valor de $diff$ é sempre ≥ 0 porque se $\text{actualRate} > \text{congWnd}/\text{RTT}_{\text{base}}$ isso significa que RTT_{base} deveria ser alterado para o último RTT medido. São também definidos dois limites para $diff$, respectivamente o limite inferior α , e o limite superior β . Quando $diff$ está abaixo de α , é praticado o incremento aditivo de `congWnd`. Quando $diff$ está acima de β , `congWnd` é reduzida de $1/8$. Quando está no meio, não se altera a dimensão da janela de saturação.

A ideia base é a seguinte: quando $diff$ está acima de β a saturação está próxima pois o débito real começa a ser inferior ao que a janela de saturação deveria permitir se o RTT estivesse ao nível de RTT_{base} , e a janela de saturação é reduzida de $1/8$. Quando $diff$ está abaixo de α , existe provavelmente a oportunidade de aumentar a

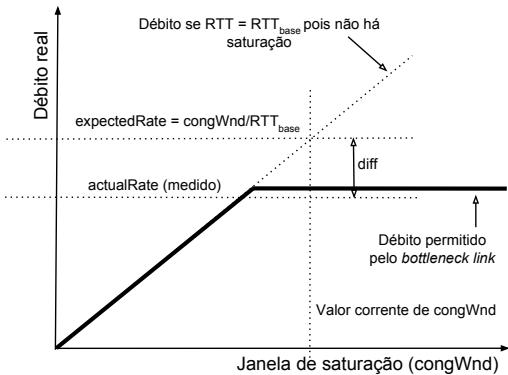


Figura 8.25: TCP Vegas: `expectedRate` e `actualRate` de uma conexão

janela de emissão e os dados em trânsito. Senão, não se altera o valor da janela pois é provável que a conexão esteja a funcionar à “velocidade de cruzeiro”.

TCP Vegas comporta-se relativamente bem mas é incapaz de competir com os derivados mais recentes de TCP Reno, obtendo uma fração inferior do *bottleneck link* pois Vegas é menos “atrevido” que Reno a aumentar a janela de saturação. Tal mostra que existe hoje em dia um problema suplementar para quem inventar um novo algoritmo de controlo da saturação e que consiste em tomar em consideração a sua coexistência com os já existentes. Algumas das ideias de TCP Vegas foram de qualquer forma retidas noutras propostas.

O débito extremo permitido pelo algoritmo de controlo da saturação Reno situa-se, em condições ideais, entre o débito disponível no *bottleneck link* para a conexão e metade do mesmo. Este resultado está relacionado com a necessidade que esse algoritmo tem de provocar saturação, para a evitar em seguida, e com o funcionamento AIMD, que usa para regular o débito.

O desempenho do algoritmo revela-se insuficiente quando as condições da rede, ou as características das conexões, estão afastadas do modelo de Internet para que foi concebido: conexões de longa duração, sobre canais com fios, com pequeno RTT e com relativamente baixo débito (alguns Mbps no máximo), e rede relativamente homogénea e com taxa de erros residual.

Muitas destas condições não se verificam hoje em dia e os algoritmos de controlo da saturação do TCP têm sofrido inúmeras melhorias para os adaptar a novos requisitos. Adicionalmente, têm sido introduzidos inúmeros refinamentos das propostas iniciais com o objectivo de melhorar o desempenho das conexões *etc.*

A dificuldade do problema advém também da utilização do mesmo protocolo em condições de utilização e de rede muito heterogéneas, abrangendo redes sem fios com taxas de erros elevadas, redes que reordenam pacotes, redes tradicionais mas com elevado RTT, conexões de pequena dimensão, redes com canais de elevado volume (débito muito elevado e RTT muito elevado).

Se o emissor conhecesse a capacidade que a rede lhe disponibiliza em cada momento, seria possível ter um desempenho superior. É o que tentou a versão TCP Vegas, que se baseia na ideia de evitar a saturação e não de a controlar. Infelizmente, não é fácil introduzir um algoritmo com uma nova filosofia caso o mesmo se revele incapaz de competir com as versões dominantes, pois isso põe em causa a equidade entre fluxos.

8.6 Resumo e referências

Resumo

Uma rede de computadores entra em saturação quando o conjunto das solicitações de encaminhamento de pacotes que recebe a conduzem a uma situação de colapso, e a quantidade real de pacotes encaminhados é inferior à sua capacidade efectiva. É uma situação análoga à que sucede nas redes viárias quando estas não suportam condições de tráfego automóvel anormais e têm lugar engarrafamentos.

Os mecanismos de controlo da saturação são mecanismos que procuram controlar dinamicamente o débito com que os pacotes são injectados na rede, tendo em consideração a situação real e a capacidade disponível, de forma a maximizarem o rendimento da rede e a entrega de pacotes aos seus destinos. As arquitecturas, protocolos e algoritmos usados para controlar as solicitações à rede, de forma a evitar que esta entre em saturação, chamam-se arquitecturas, protocolos e algoritmos de controlo da saturação.

Tirar o melhor rendimento possível da rede, sem a saturar e assegurando equidade entre os diferentes fluxos de pacotes, é um problema de optimização. Calcular a solução óptima implica dispor de uma visão global do estado da rede, dinamicamente actualizada em função da evolução das solicitações e, adicionalmente, é necessário conseguir controlar os computadores ligados à rede para controlar o seu ritmo de emissão de pacotes. Este tipo de arquitectura não é realista senão numa situação em que os gestores da rede tivessem controlo completo sobre os computadores que lhe estão ligados. No caso geral, o máximo que a rede pode esperar é alguma colaboração dos protocolos de transporte que esses computadores usam.

Geralmente o débito máximo que um fluxo de pacotes pode usar é condicionado pela capacidade disponível para o fluxo no canal mais carregado entre aqueles que o fluxo atravessa. Esse canal chama-se o canal gargalo (*bottleneck link*).

O controlo da saturação necessita de regulação do débito dos fluxos de pacotes emitidos pelos computadores ligados à rede. Este controlo pode basear-se em imposição pela rede do débito máximo, ou em *feedback* da rede aos computadores para que estes se auto-regulem. As arquitecturas baseadas em *feedback* da rede dividem-se ainda em *feedback* explícito, quando a rede dá indicações directas aos computadores para que estes se auto-regulem, e *feedback* implícito, quando são os computadores que detectam os indícios de saturação e voluntariamente se auto-regulam.

As redes TCP/IP foram definidas para utilizarem por omissão *feedback* implícito e o protocolo de transporte alvo é o TCP. Como um emissor TCP não tem noção de qual o estado da rede, nem por omissão recebe *feedback* desta, tem de experimentar ir aumentando o ritmo de emissão e, quando se aperceber que há problemas, deverá diminui-lo. A técnica de ir incrementando o ritmo de transmissão para tentar aproximar a capacidade máxima da rede sem a saturar, chama-se sondar os recursos da rede (*network resource probing*).

O protocolo TCP varia o seu ritmo de emissão máximo usando a fórmula “subida aditiva, descida multiplicativa” (*AIMD – Additive Increase, Multiplicative Decrease*). O protocolo usa a dimensão da janela de emissão para concretizar quer um mecanismo de controlo de fluxo (*flow control*), que adapta o ritmo de transmissão à capacidade do receptor consumir atempadamente os dados recebidos, quer um mecanismo de controlo

da saturação (*congestion control*), que adapta o ritmo de transmissão à capacidade disponível na rede para a conexão TCP.

O algoritmo de controlo da saturação do TCP utiliza uma variável, designada janela de saturação (*congWnd*), que limita a dimensão máxima da janela de emissão. Em cada momento, o protocolo tenta estimar a capacidade máxima disponível para a conexão no seu *bottleneck link*, subindo aditivamente o ritmo de emissão e usando os *timeouts* e os *triple duplicate ACKs* como indícios de saturação. Quando estes eventos têm lugar, a capacidade disponível é então estimada como sendo a do momento da detecção dessas ocorrências e aplica-se uma descida multiplicativa do ritmo de emissão.

Inicialmente, a janela de saturação tem a dimensão do MSS (*Maximum Segment Size*) e vai sendo duplicada em cada RTT, durante uma fase do algoritmo designada *slow start*. Quando a janela de saturação atinge um valor equivalente a metade da capacidade máxima estimada, entra numa fase chamada *congestion avoidance*, durante a qual a janela é incrementada aditivamente do valor do MSS (em bytes) em cada RTT. Na sequência de um *timeout*, a janela de saturação é reduzida de novo à dimensão do MSS e o emissor volta a entrar na fase *slow start*. Na sequência de um *triple duplicate ACK*, a janela de saturação é reduzida a metade da capacidade máxima estimada e o emissor continua na fase *congestion avoidance* até surgir de novo um *timeout* ou um evento *triple duplicate ACK*. Este algoritmo recebeu o nome de TCP Reno.

O algoritmo TCP Reno constituiu uma primeira solução para um problema complexo dada a diversidade das situações em que o controlo de saturação tem de actuar. Num caso extremo, a conexão pode atravessar uma área de rede pequena, homogénea, bem dimensionada e com RTT reduzido. No outro extremo, a conexão atravessa uma grande quantidade de canais heterogéneos, com níveis de utilização muito diversos e apresentando um RTT elevado ou até erros de transmissão, erroneamente interpretados como sinais de saturação, uma situação frequente em canais sem fios.

Em alternativa a soluções de controlo da saturação baseadas em *feedback* implícito, é possível utilizar soluções baseadas em *feedback* explícito. O mecanismo ECN – Notificação Explícita da Saturação (*Explicit Congestion Notification*) permite indicar explicitamente aos emissores TCP que devem refrear o seu débito de emissão para não saturarem a rede. Infelizmente, o mecanismo requer a cooperação entre os computadores e o nível transporte nos computadores e a sua adopção tem-se revelado lenta. Trata-se de uma implementação pragmática de *feedback* explícito, mais simples de implementar do que um mecanismo que indicasse ritmos explícitos a serem utilizados pelas diferentes conexões.

Em alternativa à notificação explícita da saturação, é possível suprimir prematuramente pacotes, na esperança que os respectivos emissores reduzam de forma equitativa o seu débito, como almeja o mecanismo de supressão prematura, aleatória, de pacotes (RED – *Random Early Detection*). Infelizmente este mecanismo, simples de concretizar, é difícil de parametrizar de forma a que o mesmo se adapte dinamicamente à situação da rede e da diversidade dos fluxos e conduza a mais proveitos que custos.

É possível demonstrar que a estratégia AIMD tende para a estabilidade e equidade na partilha de um *bottleneck link* entre várias conexões TCP. No entanto, se as conexões TCP tiverem RTTs distintos a equidade não é garantida. Por outro lado, o TCP não pode competir com fluxos de pacotes que não apliquem uma estratégia de controlo de saturação, pois reduz sempre o débito de emissão independentemente da estratégia seguida pelos seus competidores.

O débito extremo a extremo permitido pelo algoritmo de controlo da saturação Reno situa-se, em condições ideais, entre o débito disponível no *bottleneck link* para a conexão e metade do mesmo. Este resultado está relacionado com a necessidade que esse algoritmo tem de provocar saturação, para a evitar em seguida, e com o funcionamento AIMD que usa para regular o débito.

O desempenho do algoritmo revela-se insuficiente quando as condições da rede, ou as características das conexões, estão afastadas do modelo de Internet para que foi

concebido: conexões de longa duração, sobre canais com fios, com pequeno RTT e com débito relativamente baixo (alguns Mbps no máximo), e rede relativamente homogénea com taxa de erros residual.

Muitas destas condições não se verificam hoje em dia e os algoritmos de controlo da saturação do TCP têm sofrido inúmeras melhorias para os adaptar a novos requisitos. Adicionalmente, têm sido introduzidos inúmeros refinamentos da proposta inicial com o objectivo de melhorar o desempenho das conexões e o rendimento da rede. A dificuldade do problema advém também da utilização do mesmo algoritmo em condições de utilização e de rede muito heterogéneas, que vão desde redes sem fios, com taxas de erros elevadas, redes que reordenam pacotes, redes tradicionais mas com elevado RTT, conexões de pequena dimensão, redes com canais de elevado volume (débito muito elevado e RTT muito elevado), etc.

Se o emissor conhecesse a capacidade que a rede lhe disponibiliza em cada momento, seria possível ter um desempenho superior. É o que tentou a versão TCP Vegas que se baseia na ideia de evitar a saturação e não de a controlar. Infelizmente, não é fácil introduzir um algoritmo com uma nova filosofia caso o mesmo se revele incapaz de competir em pé de igualdade com as versões dominantes pois isso põe em causa a equidade entre fluxos.

Alguns dos termos introduzidos no capítulo são a seguir passados em revista. Entre parêntesis figura a tradução mais comum em língua inglesa.

Controlo da saturação (*congestion control*) Mecanismos que procuram controlar dinamicamente o débito com que os pacotes são injectados na rede, tendo em consideração a situação real e a capacidade disponível, de forma a maximizar o rendimento da rede e a entrega de pacotes aos seus destinos.

Débito útil (*goodput*) Débito real de informação entregue pela rede aos destinatários, isto é, descontando todo o desperdício introduzido pelos protocolos, quer em termos de cabeçalhos dos pacotes, quer em termos de pacotes reemitidos e recebidos em duplicado.

Poder da rede (*network power*) Quociente do débito útil (*goodput*) pelo tempo de trânsito de extremo a extremo. Numa rede saturada, o aumento do débito de injecção de pacotes na rede conduz à diminuição do seu poder pois o *goodput* diminui e o RTT aumenta.

Canal gargalo (*bottleneck link*) Canal que impõe o débito máximo disponível para um fluxo.

Sondar a rede (*Network resource probing*) Técnica que consiste em ir incrementando o ritmo de transmissão para tentar aproximar a capacidade máxima da rede, sem a saturar.

AIMD (*Additive Increase, Multiplicative Decrease*) Forma de controlo do débito de um emissor que conduz a uma utilização equitativa do canal gargalo e a uma solução com estabilidade para o problema do controlo da saturação.

Controlo da saturação com base em feedback O controlo da saturação necessita de regulação do débito dos fluxos de pacotes emitidos pelos computadores ligados à rede. Este controlo pode basear-se em imposição pela rede do débito máximo ou em *feedback* da rede aos computadores para que estes se auto-regulem.

Feedback explícito Técnica de controlo da saturação que se baseia em a rede dar indicações directas aos computadores para que estes se auto-regulem.

Feedback implícito Técnica de controlo da saturação que se baseia em os computadores detectarem os indícios de saturação. Os indícios usados são a perda de pacotes e as variações do débito útil e do RTT.

Notificação Explícita da Saturação (*ECN – Explicit Congestion Notification*) Mecanismo que permite indicar explicitamente aos emissores que devem refrear o seu débito de emissão para não saturarem a rede.

Equidade (*Fairness*) Propriedade da regulação do débito de um conjunto de fluxos que é máxima quando nenhum dos fluxos recebe melhor serviço em detrimento do serviço prestado aos fluxos com que compete. A equidade é mínima quando só um fluxo recebe serviço e os restantes nenhum.

Evitar a saturação (*Congestion avoidance*) Técnica de controlo da saturação que se baseia em evitar a saturação sem a provocar. Os indícios usados para detectar a proximidade da saturação são as variações do débito útil e do RTT.

Referências

A primeira proposta para introdução de controlo de saturação no TCP foi feita por V. Jacobson [Jacobson, 1988]. O RFC 6582 começa com um resumo onde se pode ler: “RFC 5681 documents the following four intertwined TCP congestion control algorithms: slow start, congestion avoidance, fast retransmit, and fast recovery. RFC 5681 explicitly allows certain modifications of these algorithms, including modifications that use the TCP Selective Acknowledgment (SACK) option (RFC 2883), and modifications that respond to “partial acknowledgments” (ACKs that cover new data, but not all the data outstanding when loss was detected) in the absence of SACK. This document describes a specific algorithm for responding to partial acknowledgments, referred to as “NewReno””. Este resumo testemunha que os algoritmos de controlo da saturação introduzidos inicialmente foram sendo sucessivamente melhorados de forma a incorporarem o tratamento de novas situações, não totalmente bem cobertas pelas versões anteriores.

Numa outra linha de abordagem do problema da saturação no protocolo TCP, a primeira proposta de *congestion avoidance* para o TCP foi o algoritmo Dual [Wang and Crowcroft, 1992]. O algoritmo TCP Vegas está descrito em [Brakmo et al., 1994]. Apesar de inicialmente esta abordagem não ter sido muito adoptada nas implementações usadas em produção do TCP, a investigação em torno destas propostas continuou com alguma intensidade e influenciou versões recentes dos algoritmos. O controlo da saturação com base em *feedback explícito* da rede, o mecanismo ECN, está descrito no RFC 3168. O mecanismo RED está descrito em [Floyd and Jacobson, 1993].

Com a subida da capacidade dos canais da Internet e a generalização de canais sem fios, acontecimentos que tiveram maior expressividade já no presente século, houve um renovado interesse em melhorar os algoritmos de controlo e para evitar a saturação, com o objectivo de melhorar o desempenho do protocolo TCP. Esses esforços traduziram-se em várias dezenas de propostas, algumas das quais foram introduzidas como melhoramentos dos algoritmos usados até então. As sínteses (*surveys*) [Afanashev et al., 2010; Abed et al., 2011] recenseiam muitos desses esforços. O *survey* [Chen et al., 2005] é especialmente focado nos problemas relacionados com o desempenho do TCP nas redes sem fios. Le Boudec [Boudec, 2014] apresenta uma síntese de índole mais teórica sobre o problema do controlo da saturação.

Como resultado de todos estes esforços, e tendo em consideração a impossibilidade de dispor de um único algoritmo, por muitas variantes que incorpore para lidar com todas as diferentes situações, os sistemas de operação modernos passaram a incorporar diversos algoritmos de controlo da, ou para evitar a saturação, na implementação do protocolo TCP. Por omissão, o sistema usa uma versão, mas os utilizadores, através de módulos do núcleo do sistema, de opções dos *sockets* TCP ou através de outros mecanismos de controlo do núcleo, podem alterar a versão usada. Os algoritmos disponíveis vão variando no tempo, e o leitor interessado terá de analisar o seu caso, recorrendo à documentação específica aplicável ao seu sistema.

O sistema Mac OS X, dado usar um núcleo baseado na versão de sistema Unix FreeBSD, utilizava por omissão a versão New Reno, mas é provável que incorpore actualmente diversas outras versões. No sistema de operação Linux uma versão muito usada por omissão é designada TCP Cubic [Ha et al., 2008], mas o sistema permite

hoje em dia optar entre mais de 10 versões distintas. O sistema de operação Windows incluía uma versão chamada Compound TCP [TAN et al., 2006] mas também permite a opção por outras versões. Os sistemas iOS e Android, para pequenos dispositivos móveis, usam versões especiais do núcleo FreeBSD e Linux, e provavelmente usam versões de controlo da saturação no TCP especialmente adequadas ao ambiente móvel.

Apontadores para informação na Web

- <http://ietf.org/rfc.html> – É o repositório oficial dos RFCs, nomeadamente dos citados neste capítulo.
- <http://en.wikipedia.org/wiki/Iperf> – Contém informação sobre o programa *iperf*. Este programa pode ser usado para testar o desempenho de versões dos algoritmos de controlo de saturação disponíveis. Em complemento com um emulador ou um laboratório real de redes, é particularmente útil para realizar estudos comparativos.
- <https://www.wireshark.org> – Contém informação sobre o *wireshark*. Este programa é um monitor dos pacotes que atravessam uma interface de rede de um computador. Como pode registar todos os dados contidos nos cabeçalhos TCP, permite a partir dos mesmos representar graficamente a evolução da janela e outros dados do cabeçalho.
- <http://www.web10g.org> – Contém informação sobre o módulo *Web10G*. Este módulo, para o núcleo do sistema Linux, permite aceder a todas as variáveis internas usadas pela implementação do TCP.
- <http://www.linuxfoundation.org> – Permite aceder à documentação do módulo *tcpprobe* para Linux. É um módulo que fornece dados como o módulo *Web10G*, mas de forma mais sintética.

8.7 Questões para revisão e estudo

1. Verdade ou mentira? Justifique a sua resposta.
 - (a) Durante a fase *congestion avoidance* a janela do emissor é aumentada de 1 MSS sempre que é recebido um ACK.
 - (b) Durante a fase *congestion avoidance* a janela do emissor é aumentada de 1 MSS sempre que são recebidos todos os ACKs correspondentes aos segmentos da janela.
 - (c) O protocolo TCP evita a saturação da rede não variando o tamanho da janela de emissão.
 - (d) Durante a fase *slow start* a janela é aumentada de 1 MSS sempre que são recebidos todos os ACKs correspondentes aos segmentos da janela.
 - (e) Durante a fase *slow start* a janela é aumentada de 1 MSS sempre que é recebido um ACK.
 - (f) Um emissor TCP só usa segmentos de dimensão MSS. O receptor tem um *buffer* de recepção de dimensão MSS bytes. Logo, a janela do emissor nunca ultrapassa o valor de MSS bytes.
 - (g) Um emissor TCP só usa segmentos de dimensão MSS. O receptor tem um *buffer* de recepção de dimensão MSS bytes. O algoritmo Reno comportar-se neste caso da mesma forma que o algoritmo Tahoe.
 - (h) O algoritmo de controlo de saturação usado por um emissor TCP depende do algoritmo de controlo de saturação usado pelo receptor.

- (i) Admitindo por hipótese que uma conexão TCP apenas atravessa canais de muito alto débito que não perdem pacotes por erros de transmissão, a versão Reno do algoritmo de controlo da saturação é adequada mesmo que o RTT seja de 100 ms.
- (j) O controlo da saturação tem uma solução completamente adequada através do algoritmo Reno sejam quais forem as condições de funcionamento da rede.
2. n computadores, cada um com uma única conexão TCP, partilham o mesmo *bottleneck link*. Não existe outro tráfego no mesmo canal. Qual é o resultado de um *browser* HTTP num dos n computadores pessoais abrir k conexões em paralelo para vários servidores distintos? Exprima o resultado em termos de n e k .
3. A Figura 8.26 representa a evolução da janela do emissor de uma conexão TCP ao longo do tempo. O eixo das ordenadas representa o tamanho da `congWnd` (a unidade é o MSS) e o das abcissas o tempo (a unidade é o RTT). O emissor está a usar o algoritmo de controlo da saturação Tahoe ou Reno? Justifique.

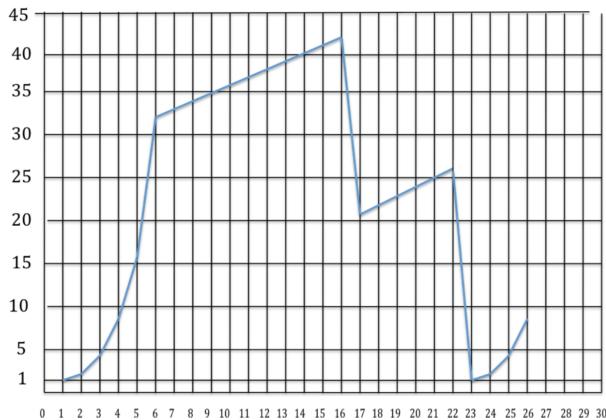


Figura 8.26: Evolução da dimensão da janela do emissor de uma conexão TCP

4. Um *browser* HTTP necessita de obter um objecto com 4.000 bytes de um servidor através de uma conexão TCP. Os canais que ligam o cliente e o servidor HTTP têm uma capacidade superior a 100 Mbps. O RTT entre os dois computadores é de 200 ms e a conexão TCP usa um MSS de 1250 bytes. A janela inicial de uma conexão é da dimensão de MSS bytes. Qual é aproximadamente o tempo necessário ao *browser* para obter o objecto sabendo que tem de abrir a conexão TCP e enviar uma primeira mensagem com o pedido?
5. Explique porque razão a capacidade do *bottleneck link* não é dividida equitativamente entre várias conexões TCP com RTTs distintos. Em particular, explique porque razão as conexões com menor RTT obtêm uma fração superior da capacidade do canal.
6. Qual o débito máximo de uma única conexão TCP que atravessa um conjunto de canais com o débito de 200 Mbps, apenas carregados a 10% com outro tráfego, com um RTT de extremo a extremo de 100 ms, mas em que ambas as partes não podem usar a opção “*receiver window scaling*”?
7. Dentro da rede dois comutadores estão interligados por dois canais *full-duplex* distintos, um com o tempo de propagação extremo a extremo de 50 ms e o outro

com o tempo de 100 ms. Para dividir a carga, os segmentos de uma conexão TCP que passa por esses dois comutadores são enviados alternativamente, ora por um, ora pelo outro canal. Qual o resultado final deste melhoramento em termos do desempenho da conexão TCP?

8. Um canal com a capacidade de 100 Mbps é atravessado por 10 fluxos de pacotes e é o *bottleneck link* desses fluxos. Cinco dos fluxos correspondem a conexões TCP usadas para transferir ficheiros e podem, cada uma delas, ocupar todo o canal, isto é, potencialmente usar os 100 Mbps. Por hipótese, as conexões TCP têm RTTs semelhantes. Os outros 5 fluxos são de pacotes UDP pertencentes a 5 fluxos de vídeo independentes, cada um dos quais requer 2 Mbps de capacidade. Como é dividida aproximadamente a capacidade do canal pelos 10 fluxos?
 - (a) Todos os 10 fluxos ficam aproximadamente com 2 Mbps cada um;
 - (b) Todos os 10 fluxos ficam aproximadamente com 10 Mbps cada um;
 - (c) Os fluxos UDP ficam com aproximadamente 1 Mbps cada um e os fluxos TCP ficam com 10 Mbps cada um;
 - (d) Os fluxos UDP ficam com aproximadamente 2 Mbps cada um e os fluxos TCP ficam com 18 Mbps cada um;
 - (e) Os fluxos UDP ficam com aproximadamente 2 Mbps cada um e os fluxos TCP ficam com 10 Mbps cada um;
 - (f) Todos os 10 fluxos ficam aproximadamente com 100 Mbps cada um;
 - (g) Nenhuma destas opções.
9. Um computador A está a transmitir um ficheiro de grande dimensão para um computador B através de uma conexão TCP. Ambos os computadores usam a versão Reno do algoritmo de controlo da saturação. Indique, de forma justificada, qual débito médio aproximado entre A e B nas condições indicadas a seguir.
 - (a) A e B estão ligados através de um canal directo *full-duplex*, dedicado exclusivamente à transferência, com uma velocidade de transmissão de V_t bps e um tempo de propagação desprezável.
 - (b) A e B estão ligados à Internet através de canais com capacidade de V_t bps. A janela de recepção máxima do *socket* de B tem J bits, com $J \gg MSS$ da conexão, o tempo de ida e volta entre A e B são RTT segundos, $V_t \gg J/RTT$ e verifica-se que nunca se perdem segmentos entre A e B.
 - (c) A e B estão ligados à Internet através de canais com capacidade de V_t bps. No interior da rede o *bottleneck link* só disponibiliza em média $V_{bottleneck}$ bps para a conexão, $V_t \gg V_{bottleneck}$, não existem limites devidos à dimensão da janela do receptor e os únicos segmentos que se perdem são retransmitidos por *Fast Retransmit*.
10. Um programa criou uma conexão TCP entre os computadores A e B com o MSS de 10.000 bits. Essa conexão é a única actividade de rede em ambos os computadores. A funciona como emissor e B como receptor de um ficheiro de grande dimensão. Os canais que ligam A e B à rede têm o débito de 5 Mbps. O RTT entre A e B é de 100 ms. Não existem limites ao débito da conexão devidos a factores aplicacionais ou de dimensão dos *buffers* de A ou B.
 - (a) Qual o menor valor, em segmentos, da janela de emissão que maximiza a taxa de utilização do canal que liga A à rede?

- (b) Admita que o *bottleneck link* entre A e B apenas disponibiliza 1 Mbps para a conexão. Qual o valor médio aproximado, em MSSs, da janela de emissão no computador A, admitindo que os únicos segmentos que se perdem são retransmitidos por *Fast Retransmit*. Ambos os computadores usam a versão Reno do algoritmo de controlo da saturação.
11. O computador A abriu uma conexão TCP para o computador B com um MSS de 10.000 bits (1250 bytes aproximadamente). Os dois computadores estão ligados numa rede com o RTT de 98 ms. A rede tem uma grande capacidade interna e não perde pacotes. O computador A está a transmitir continuamente dados para B e a transferência não é refreada por factores ligados à aplicação. Os computadores A e B estão ligados à rede por canais que funcionam a 10 Mbps. O computador B tem uma janela máxima de recepção (*receiving window* máxima) de 100.000 bits (cerca de 12500 bytes). A janela inicial de uma conexão é da dimensão de MSS bytes.
- (a) Qual a dimensão máxima atingida pela janela de emissão de A em múltiplos do MSS?
 - (b) Quantos RTTs são necessários para que A atinja essa janela de emissão quando arranca na fase *slow start* do protocolo?
 - (c) Quando a janela de emissão de A estabilizar, qual a velocidade média a que a transferência do ficheiro de A para B se processa?
12. Um cliente TCP necessita de obter 5 objectos de um servidor. Cada objecto tem 5 K Bytes. O cliente e o servidor estão separados por um tempo de trânsito de cerca de 50 ms pelo que o RTT entre ambos é de cerca de 100 ms. As conexões TCP não se encontram limitadas nem pela velocidade de transmissão dos canais que ligam os computadores à rede, nem pelo espaço livre nas janelas de recepção dos receptores, nem pela capacidade da rede. O MSS é de 1250 bytes e todas as conexões iniciam a gestão da janela de emissão pela fase *slow start*. A janela inicial de uma conexão é da dimensão de MSS bytes. Calcule o tempo necessário para obter os 5 objetos quando o cliente usa uma só conexão TCP e quando usa 5 conexões TCP abertas em paralelo. Justifique a sua resposta.

Capítulo 9

Transporte de dados multimédia

There are no facts, only interpretations.

– Autor: *Friedrich Nietzsche (1844-1900)*

A informação transmitida através de uma rede digital de computadores corresponde a sequências de símbolos, codificados como sequências de bits, que vão ser posteriormente processados pelos destinatários, *i.e.*, por máquinas ou por humanos (pelo menos por agora são poucos os outros animais que usam redes de comunicação). Quando o destinatário final são os humanos, a sequência de símbolos é transformada num conjunto de impulsos sensoriais de forma a poder ser absorvida e interpretada pelo destinatário. Por exemplo, os símbolos contidos numa mensagem são visualizados num ecrã de computador, ou são traduzidos por um altifalante num conjunto de sons, ou num conjunto de imagens animadas que reproduzem um filme. Os sons são percepcionados através do ouvido e as imagens através da vista.

No caso particular dos sons e das imagens, os humanos têm, em muitas situações, a capacidade de obter, grosso modo, a mesma quantidade de informação independentemente de os símbolos recebidos serem diferentes dos emitidos, pois quer o ouvido, quer a vista, têm capacidade de reconhecer o significado de sons e imagens degradadas face à sua versão inicial. Apenas a qualidade e o conforto do destinatário ao processar essa informação é que vai variar. Por outro lado, no caso dos filmes, mas também no caso dos sons, a interpretação da informação recebida está dependente da que foi recebida anteriormente, pois essa informação só adquire o seu significado inserida num fluxo de informação. Por exemplo, uma imagem de um filme só se comprehende completamente quando inserida na sequência de imagens que a precederam.

Existem portanto situações em que a informação transmitida pela rede corresponde a fluxos de informação que codificam sequências de sons e imagens, ou de forma mais rigorosa, em que a informação transmitida é multi-formato (imagens, sons, legendas, *etc.*) e organizada em fluxos sincronizados (*e.g.*, os sons associados a imagens têm de estar sincronizados com estas). O termo popular para este tipo de informação é **informação multimédia em fluxos** ou simplesmente **informação multimédia** (*multimedia streaming* ou simplesmente *multimedia*). Os termos populares são um pouco enganadores porque qualquer ficheiro pode ser um ficheiro multimédia por ter palavras, frases, parágrafos e imagens, mas nós vamos utilizá-los também daqui para a frente.

Enquanto que a transmissão indiscriminada de informação tem de ser feita de forma fiável, pois não se sabe qual o seu tipo e a utilização que dela vai ser feita

pelo destinatário, a informação multimédia pode ser transmitida de forma não fiável. A solução não coloca problemas desde que o resultado seja de “qualidade aceitável” para o destinatário, sendo o adjetivo “aceitável” algo que depende do contexto e das exigências do utilizador.

Postas as coisas assim, parece que é suficiente enviar a informação multimédia através de um protocolo sem recuperação de erros, como o UDP por exemplo. Bom, as coisas não são assim tão simples, pois para que o resultado final seja de “qualidade aceitável” é necessário impor limites à taxa de erros mas também ao tempo de transferência de extremo a extremo e ao *jitter*. Por outro lado, é também possível tentar adaptar a resolução da informação transmitida à capacidade disponível na rede, o que dá uma nova perspectiva sobre o controlo de saturação.

No caso particular da transmissão de fluxos de informação multimédia podem ser usadas técnicas de transmissão de dados com perda de resolução, que tomam em consideração as características particulares da aplicação final a que os mesmos se destinam. Trata-se de um caso particular em que existe uma interligação entre o transporte e as aplicações. De alguma forma é mais um caso em que parece que a separação entre camadas não é absoluta e por essa razão este capítulo envolve aspectos que ultrapassam meramente a problemática do transporte de dados.

O capítulo começa exactamente por apresentar uma panorâmica breve sobre como são codificados de forma digital o som e a imagem, e de como essas informações são organizadas em fluxos sincronizados. A seguir apresentam-se os requisitos típicos das aplicações multimédia, e de que forma estes influenciam os protocolos e as soluções aplicacionais usadas pelas mesmas. Segue-se uma secção sobre as técnicas de correção de erros com e sem perda de resolução, usadas pelos canais lógicos extremo a extremo que suportam essas aplicações. Finalmente, é apresentado o protocolo *RTP – Real Time Transport Protocol*, que complementa o protocolo UDP com os mecanismos necessários para transmitir fluxos de informação multimédia em tempo real.

9.1 Codificação de informação multimédia

Codificação do som

Os objectos que vibram emitem vibrações, *i.e.*, ondas mecânicas, que se propagam pela atmosfera e que quando atingem as membranas e os ossos do ouvido dos animais são percepcionados como sons. O ouvido humano só consegue detectar sons compostos por ondas mecânicas cujas frequências de oscilação estejam entre 20 e 20.000 Hz¹ mas outros animais conseguem percepcionar frequências numa gama diferente. Aliás, a gama audível dos humanos varia de pessoa para pessoa e raramente é tão alargada como a indicada. A voz humana é composta de ondas mecânicas emitidas pelas cordas vocais numa gama de frequências grosso modo entre 300 e 8.000 Hz.

Um microfone é um dispositivo que transforma uma onda sonora num sinal eléctrico cuja variação é semelhante à da onda sonora, e um altifalante é um dispositivo que transforma um sinal eléctrico que segue o andamento de uma onda sonora numa onda sonora semelhante.

O sistema telefónico analógico tradicional realizava chamadas telefónicas à distância através da transmissão das ondas sonoras na forma de sinais eléctricos cuja variação era análoga à da voz dos interlocutores. Posteriormente, o sistema telefónico analógico deu lugar ao sistema telefónico digital que passou a transmitir o som digitalizado, pelo menos entre as centrais telefónicas, ver a Figura 9.1. Esta forma de codificação do

¹A frequência dos fenómenos oscilatórios periódicos mede-se em Hertz (Hz) e 1 Hz corresponde à frequência de uma oscilação por segundo. Uma onda sinusoidal de 1 Hz corresponde a uma sinusoidal que completa o seu período em 1 segundo.

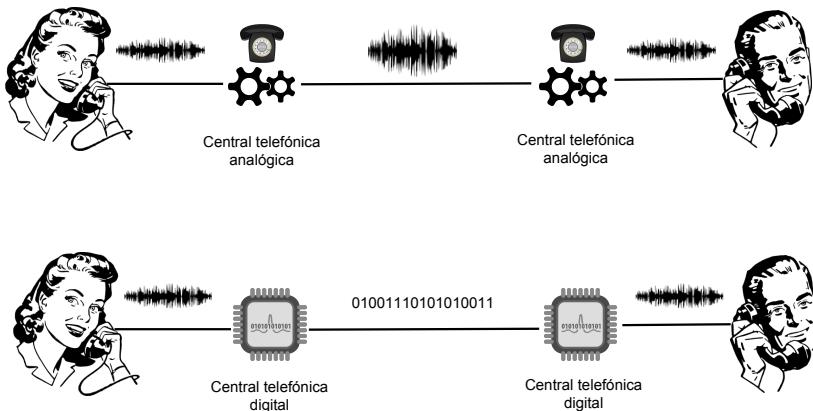


Figura 9.1: Transmissão do som à distância com base num sinal eléctrico analógico e com base no sinal digitalizado

som é também a adequada à sua transmissão por uma rede de pacotes de dados, visto que esta é capaz de transmitir sequências de valores.

A digitalização de uma onda (sonora, eléctrica, *etc.*) consiste em produzir uma sequência de valores numéricos que têm o mesmo andamento da onda. Os dispositivos que realizam esta transformação chamam-se **conversores analógico-digital (ADC - Analog to Digital Converter)**. A conversão inversa é feita por **conversores digital-analógico (DAC - Digital to Analog Converter)**.

Cada valor numérico diz-se uma **amostra (sample)**, e a sequência de valores corresponde a uma sequência de amostras tomadas a intervalos regulares. A periodicidade, ou seja o intervalo de tempo que separa cada amostra, fica caracterizada pela **frequência de amostragem (sampling rate)**.

A fidelidade entre a sequência de valores numéricos e a onda depende de dois factores: a frequência de amostragem e a resolução com que são expressos os valores numéricos. Quanto maior a frequência e quanto maior a resolução mais fidedigna (mais fiel) é a digitalização com respeito à onda original analógica e maior será também a quantidade de informação contida na sequência de valores numéricos.

Qual deve ser então a frequência de amostragem e a resolução que deve ser usada? A resposta a esta questão é dada pela teoria do processamento dos sinais digitais e depende do contexto de utilização. Segundo esta teoria, um sinal (sonoro, eléctrico, *etc.*) é equivalente à soma de um conjunto de sinais sinusoidais de diferentes frequências e amplitudes. Segundo a mesma teoria, se a frequência da componente de mais alta frequência for f , uma frequência de amostragem superior a $2f$ não acrescenta mais informação e portanto não aumenta a fidelidade.

Assim, qualquer som audível fica correctamente digitalizado do ponto de vista das frequências nele presentes desde que a frequência de amostragem seja superior a 40 KHz (40.000 vezes por segundo) pois o ouvido humano não percepção sons cuja frequência seja superior a 20.000 Hz. A voz humana usa um conjunto de frequências

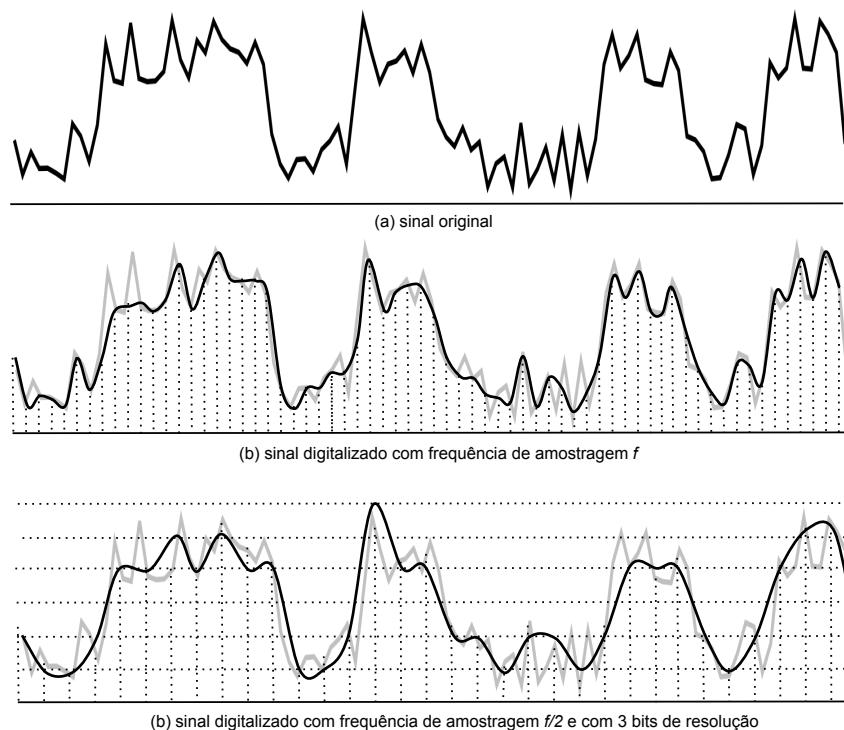


Figura 9.2: Impacto da frequência de amostragem e da resolução no sinal digitalizado

mais restritas que o conjunto dos sons audíveis e a experiência mostra que para efeitos de uma chamada telefónica, basta considerar frequências até 4 KHz, do que resulta uma frequência de amostragem de 8 KHz. Para a qualidade da voz característica dos telemóveis primitivos pode até usar-se uma frequência mais baixa, pois quanto menor for a frequência de amostragem, menor será a quantidade de informação produzida por unidade de tempo.

A resolução dos valores digitalizados está dependente do número de bits usados para representar cada valor da amostra. A deformação do sinal digitalizado em função da frequência de amostragem e da resolução é ilustrada pela figura 9.2.

Os valores normalizados inicialmente pelas indústrias de telecomunicações e da música foram respectivamente 8 KHz de frequência de amostragem e 8 bits de resolução para a voz no sistema telefónico digital, e 44,1 KHz de frequência de amostragem e 16 bits de resolução para o som nos CDs áudio. A opção no que diz respeito ao sistema telefónico introduz distorção em frequência na voz, pelo que a qualidade final do som pode não ser suficiente para transmitir uma ária de ópera pelo telefone convencional, visto que a forma de digitalização usada filtra as frequências superiores a 4 KHz.

Ambas as formas de digitalização introduzem distorção quantitativa pois o ouvido humano pode, no limite, distinguir intensidades distintas cuja gama de intensidade relativa pode ir de 1 a 1.000.000, enquanto que para a voz telefónica apenas são usados 256 níveis de intensidade distintos, e nos CDs musicais apenas 65.536 níveis. De qualquer forma essas resoluções têm sido consideradas suficientes para as conversas telefónicas e para a reprodução de música em alta fidelidade (excepto por alguns audiófilos mais exigentes).

Assim, a quantidade de informação por unidade de tempo que é produzida por uma chamada telefónica é de 64 Kbps (8 bits \times 8 KHz) e cada canal estéreo de música em alta fidelidade produz 705,6 Kbps (16 bits \times 44,1 KHz) ou 1,4112 Mbps para os dois canais. Estes valores correspondem aos débitos mínimos extremo a extremo necessários para transmitir essa informação por uma rede de computadores se não for usada compressão. Na realidade os débitos necessários são superiores, quer devido ao espaço ocupado pelos cabeçalhos, quer devido à necessidade de compensar o *jitter*.

Dados os débitos envolvidos, existem motivações significativas para usar compressão do som de forma a poupar capacidade na rede. Os algoritmos usados para comprimir o som são bastante diferentes dos algoritmos gerais de compressão, pois no caso do som os algoritmos de compressão usados podem conduzir a perda de informação. Um algoritmo de compressão que conserva a quantidade de informação após a descompressão chama-se um algoritmo sem perdas (*lossless compression*), enquanto que um algoritmo que conduz a perdas após a descompressão, chama-se naturalmente um algoritmo com perdas (*lossy compression*).

Os algoritmos de compressão do som mais eficientes são algoritmos com perdas, mas têm sido estudados intensivamente de modo a tornar essas perdas tão imperceptíveis quanto possível pelos humanos e por isso dizem-se psico-acústicos. Com efeito, o ouvido humano tem uma capacidade muito elevada de reconhecer frequências e intensidades distintas dos sons, mas não ao mesmo tempo. Assim, a presença num som de certas frequências audíveis cancela a hipótese de o ouvinte distinguir outras que lhe estejam próximas, e a capacidade de distinguir certas frequências do som depende também da intensidade de outros sons que lhes estejam próximos e presentes em simultâneo.

Os algoritmos de compressão usados exploram estas propriedades do ouvido humano para obterem factores de compressão muito significativos. Por exemplo, a voz telefónica pode ser transmitida com qualidade aceitável a 64 Kbps usando a forma de digitalização apresentada acima (8 KHz de frequência de amostragem e 8 bits de resolução). Esta forma de codificação digital costuma designar-se por PCM (*pulse code modulation*) e foi normalizada pela ITU através da norma G.711 em 1972. A norma G.711 produz uma amostra de som digitalizado de 125 em 125 microssegundos. No entanto, é também possível transmitir voz com um nível de qualidade aceitável para conversas telefónicas usando a norma G.729, que usa um algoritmo de compressão com perda de resolução mas que requer apenas o débito de 8 Kbps.

A música é transmitida usando algoritmos de compressão que se têm tornado populares entre o grande público, nomeadamente o MP3 (*MPEG-1, 2, 2.5 audio layer III*) e o AAC (*MPEG-2, 2.5, 4 Advanced Audio Coding*) e muitos outros, alguns normalizados e públicos e outros proprietários.

Por exemplo, os formatos MP3 e AAC conseguem qualidade de som variável e podem usar uma gama também variável de débito por canal: de 8 a 320 Kbps para o MP3 e de 8 a 529 Kbps para o AAC. Nos débitos maiores a compressão pode ser sem perdas. Em geral é possível obter uma qualidade aceitável com 96 Kbps por canal, mas para obter uma qualidade já próxima do CD original, dependendo do formato, é necessário usar resoluções maiores.

Um outro aspecto importante a reter é que estes algoritmos de compressão necessitam de actuar sobre quantidades mínimas de som para conseguir realizar a compressão e por isso introduzem atrasos suplementares. Assim, o MP3 actua sobre um mínimo de 100 ms de som digitalizado e o AAC em certas resoluções ainda pode necessitar de períodos de tempo maiores. Estes atrasos são independentes da capacidade de processamento e dos atrasos de extremo a extremo da rede, somando-se aos mesmos.

O som digitalizado consiste numa sequência de valores numéricos que representam a intensidade da onda sonora amostrada (medida) a intervalos regulares. A resolução da digitalização corresponde ao número de bits usados para representar os valores medidos.

A frequência de amostragem determina as frequências presentes na representação digital. É normal usar 8 bits de resolução e 8 KHz de frequência de amostragem para a digitalização da voz nos sistemas telefónicos, ou 16 bits / 44,1 KHz para a música, ou ainda maior resolução e frequência de amostragem quando existem requisitos de muito alta fidelidade.

Devido às características percepcionais do sentido da audição é possível aplicar algoritmos de compressão do sinal sonoro digitalizado. Estes algoritmos podem implicar perda de informação (*são lossy*) mas não perda de significado nos sistemas telefónicos, nem diferenças demasiado perceptíveis da qualidade musical. Os ganhos assim obtidos podem reduzir o débito do fluxo digital correspondente ao som digitalizado até uma ordem de grandeza, sem um impacto significativo na utilidade final da informação sonora transmitida.

Codificação das imagens

As imagens digitais, mesmo aquelas que parecem contínuas, são apresentadas pelos ecrãs digitais como um conjunto de pixels, *i.e.*, um conjunto de pontos discretos, ou unidades de informação elementares sobre a cor e intensidade da imagem². A vista humana é capaz de reconhecer a presença dos diferentes pixels em imagens digitais de baixa qualidade, ou seja, com pouca densidade de pixels por unidade de superfície. A partir do momento em que a densidade de pixels por unidade de superfície vai aumentando, a vista humana vai sendo incapaz de distinguir os pixels uns dos outros e a qualidade da imagem vai aumentando, passando o utilizador a vê-la como um contínuo.

Para representar imagens a preto e branco basta associar a cada pixel a intensidade da luz branca. Dependendo da resolução de cada pixel, pode usar-se 1 bit por pixel (o pixel está a preto ou a branco) ou vários bits onde se representam várias intensidades de luz branca. Já para representar imagens a cores é necessário associar também uma cor a cada pixel. Um forma comum de obter as diferentes gamas de cores mistura nas quantidades adequadas as cores vermelho, verde e azul (forma de codificação designada como *RGB – Red Green Blue*), ver a Figura 9.3.

Assim, numa imagem a cores, a cada pixel está associada uma quantidade de informação que codifica a intensidade de cada uma destas três cores. Em geral usam-se dois bytes (*High Color*) ou três bytes (*True Color*) para representar cada pixel. Quando se usam 2 bytes (16 bits) são usados 5 bits para o vermelho, 5 bits para o azul e 6 bits para o verde pois a vista é mais sensível ao verde. No caso da representação com 3 bytes usa-se 1 byte por cor para representar a intensidade da presença da mesma. Esta forma de codificação é conhecida por RGB888. É também frequente encontrar a forma de representação ARGB888 que acrescenta 1 byte à representação RGB888 para codificar o valor da transparência (*alpha value*), resultando em 4 bytes por pixel (32 bits).

As imagens de muito alta qualidade nos ecrãs de computador actuais são conseguidas com a codificação ARGB888 e uma densidade de pixels superior a 8.000 pixels / cm^2 . Uma imagem de televisão 4K contém 4.096×2160 pixels (cerca de 8.000.000 milhões de pixels no ecrã) codificados em RGB888.

²A palavra pixel é um neologismo derivado do mesmo termo em inglês, que é formado a partir da união de dois elementos: “pix” (de *picture*) e “el” (de *element*).

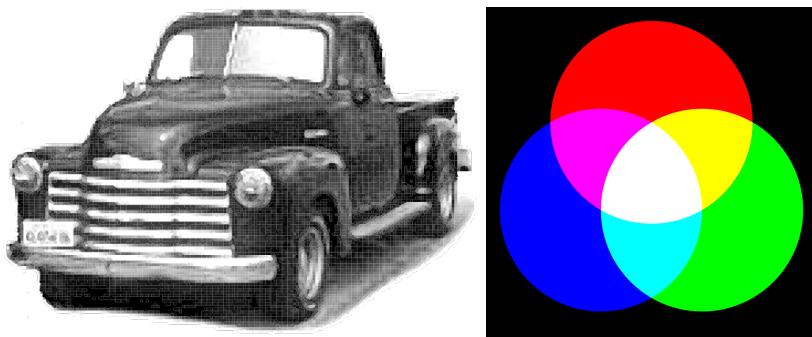


Figura 9.3: Representação de uma imagem usando pixels e o sistema de cores RGB

Um filme é apresentado fazendo desfilar a um ritmo adequado um conjunto de imagens em sucessão. O termo em português é uma sucessão de fotogramas. O termo em inglês para o número de fotogramas por unidade de tempo é *frame rate* (cada imagem é designada por *image frame* ou simplesmente *frame*). A vista humana é incapaz de distinguir umas imagens das outras a partir de um certo *frame rate*, nomeadamente cerca de 30 imagens por segundo. Actualmente a indústria cinematográfica usa 24 ou 48 imagens por segundo e a televisão usa 25 ou 50 (Europa) e 30 ou 60 (EUA). A tendência é para os valores mais altos serem os adoptados pois permitem melhor qualidade.

Daqui resulta que a quantidade de informação necessária para transmitir de forma digital um filme, sem compressão, mas com qualidade, é muitíssimo elevada. Por exemplo, com 50 imagens por segundo de 8.000.000 pixels cada, usando 3 bytes para representar cada pixel, a quantidade de informação por unidade de tempo seria: $50 \times 8 \cdot 10^6 \times 24 \text{ bps} = 9,6 \times 10^9 \text{ bps}$ ou 9,6 Gbps. Mesmo usando uma resolução inferior, a quantidade de informação atinge muito facilmente a gama das centenas de mega (ou mesmo dos giga) bits por segundo.

Felizmente que, tal como o som, também as imagens podem ser comprimidas de forma muito significativa, com perda de resolução mas eventualmente sem que o observador se aperceba disso. Assim, as imagens fixas podem ser comprimidas usando várias técnicas relativamente simples de descrever (mas não necessariamente de implementar).

A primeira é alterando a resolução e fundindo um conjunto de pixels vizinhos num só e suprimindo detalhes que a vista não distingue normalmente (sem lupa!). Na Figura 9.4 a técnica é ilustrada mas com perda de resolução perceptível ao utilizador. Frequentemente esta técnica de compressão é imperceptível à vista humana desde que a dimensão da imagem seja a adequada. No entanto, caso a imagem comprimida veja a sua dimensão ampliada, a densidade de pixels por unidade de superfície altera-se e as imperfeições aparecem, pois a ampliação é feita por interpolação. É por esta razão que as fotografias de qualidade média não resistem a serem impressas como *posters*. Uma segunda técnica consiste em fundir pixels contíguos com cores e intensidade semelhantes, como por exemplo a zona de uma imagem que contém um fundo de cor constante, ou seja explorando a redundância espacial.

O formato normalizado de representação de imagens fotográficas mais comum é o JPEG (Joint Photographic Experts Group) que especifica um formato comprimido de representação de imagens fixas. A norma recorre a um algoritmo de compressão que



Figura 9.4: Compressão de uma imagem alterando a resolução (logo do IDE Eclipse)

incorpora várias técnicas de compressão, com e sem perda de resolução.

É possível representar um filme mostrando em sucessão várias imagens independentes representadas no formato JPEG. Como é evidente, é possível fazer muito melhor que isso pois na grande maioria dos filmes muitos fotogramas (*image frames*) estão correlacionados e só apresentam pequenas modificações em relação aos precedentes, ver um exemplo na Figura 9.5.

O formato mais popular para representação com compressão de filmes é o formato MPEG (Motion Picture Experts Group). A norma MPEG tem vários formatos possíveis que contemplam para além das imagens, um ou mais canais de audio e canais com as legendas, todos sincronizados com as imagens.

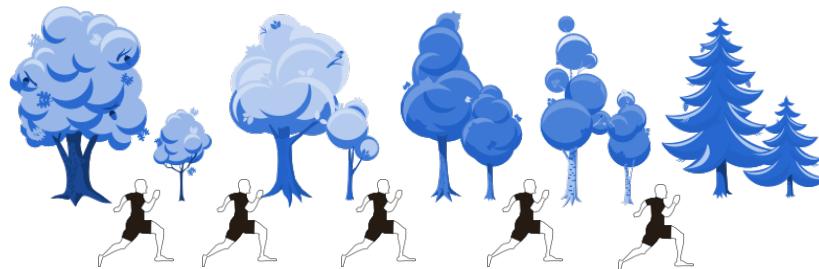


Figura 9.5: Compressão de um filme explorando a redundância entre fotogramas – o fundo é comum aos diferentes *frames*, só o corredor difere de fotograma para fotograma

A variante MPEG-1 corresponde à qualidade da televisão analógica tradicional e produz um débito médio entre 1 e 3 Mbps, dependendo da qualidade seleccionada. Os formatos de vídeo do MPEG-1 produzem um fluxo de tipo CBR (*Constant Bit Rate*) pois a forma de compressão usada não depende das características da imagem e do movimento. O formato MPEG-2 corresponde à qualidade dos DVDs e, dado o tipo de compressão usada, conduz a um fluxo de tipo VBR (*Variable Bit Rate*) dependendo do movimento e das características das cenas com um débito médio de 4 Mbps. A variante MPEG-4 permite vários formatos e resoluções entre as quais a resolução da televisão de alta definição e a dos discos Blu-ray e o débito médio depende do formato usado. Na difusão satélite digital são usados os formatos MPEG-2 e -4.

Se todos os fotogramas MPEG representassem apenas as modificações com respeito

ao precedente, não era possível ver o filme senão começando pelo início e indo até ao fim, e não seria possível avançar nem recuar a imagem por exemplo. Por outro lado, também não seria possível perder um único *frame* caso estes fossem transmitidos pela rede. Finalmente, verifica-se também que a maioria dos filmes mudam de cena com um intervalo de alguns segundos (3 a 10 segundos por exemplo).

Por estas diferentes razões o formato MPEG especifica diferentes tipos de *frames*: os I-*frames* que contém imagens completas e capazes de serem processadas individual e isoladamente, e os P-*frames* e os B-*frames* que são *frames* que apenas contém diferenças relativamente aos precedentes e aos vizinhos (do passado ou do futuro). A descodificação dos I-*frames* é semelhante à descodificação de imagens JPEG, mas a descodificação dos outros tipos de *frames* exige a utilização de um *buffer* com vários *frames*. Todas estas técnicas mostram que a descodificação MPEG requer um *playout delay* suplementar.

Uma imagem digitalizada contém uma sequência de valores que codificam a cor e a intensidade de um conjunto de pixels, *i.e.*, de unidades elementares da imagem. O número de bits usados para codificar o valor de cada pixel e a densidade de pixels da imagem condicionam a sua qualidade. Um filme é uma sequência de imagens justapostas (*image frames*), com frequências entre 20 e 60 imagens por segundo.

O débito de informação por unidade de tempo requerido para a transmissão de imagens fixas e de filmes é muito elevado se não se utilizarem algoritmos de compressão. Dadas as características de redundância das imagens e dos filmes, assim como as características da vista humana, é possível utilizar algoritmos de compressão de imagem e de filmes que reduzem esse débito até 3 ordens de grandeza através de perda de informação, mas muitas vezes com resultados imperceptíveis para a vista.

Noção de codec

Quando informação multimédia é transmitida através de uma rede, é necessário que os emissores e os receptores estejam de acordo sobre o formato e a resolução usados para interpretar o conteúdo recebido.

Um codec é um dispositivo software, hardware ou misto, capaz de codificar e descodificar (COder / DECoder) informação multimédia de acordo com um formato específico. Emissores e receptores do mesmo fluxo multimédia têm de usar codecs equivalentes.

Para melhorar a modularidade dos sistemas hardware / software usados para transmissão e interpretação de informação multimédia (emissores e receptores satélite, emissores e receptores de televisão digital terrestre, aparelhos de leitura de DVD ou outros discos, software multimédia, *etc.*) estes podem usar ou ser estendidos com diferentes codecs. Estas operações consistem simplesmente em permitir que os diferentes dispositivos usem diferentes normas e formatos de informação multimédia. Assim, muitas vezes confunde-se o dispositivo codec com o próprio formato.

Os codecs que utilizam formatos normalizados obedecem a normas bem conhecidas e cuja especificação é pública. Os codecs que utilizam formatos não normalizados podem usar formatos proprietários e secretos (como por exemplo os codecs usados pela aplicação Skype) ou formatos públicos não normalizados.

Alguns dos codecs mais populares são da família MPEG, estão implementados em hardware e estão integrados em todas as televisões digitais modernas, DVDs, ou ainda na maioria dos softwares multimédia actuais.

A informação multimédia tem características particulares que permitem que a “mesma” informação seja transmitida com resoluções distintas, ou mesmo com erros, sem que a sua utilidade seja afectada na mesma proporção.

Esta característica, a par de outros requisitos das aplicações que usam informação multimédia, tem um impacto significativo sobre as técnicas de transmissão de informação multimédia usadas sobre uma rede de pacotes.

No resto do capítulo utilizaremos informalmente o termo codec para designar quer o dispositivo, quer a norma e os formatos que este implementa.

9.2 Transporte sobre TCP e sobre UDP

Existem diversos tipos de aplicações multimédia que usam redes de computadores. No primeiro grupo podemos incluir as aplicações que reproduzem ou visualizam³ de forma diferida informação multimédia a partir de um ficheiro. Estas aplicações não impõem nenhum requisito novo ao transporte de informação pois os ficheiros multimédia podem ser transmitidos usando o protocolo TCP por exemplo. Os exemplos mais comuns deste tipo de aplicações são aquelas que permitem visualizar filmes ou tocar música a partir de ficheiros registados no disco local dos computadores.

A novidade é introduzida pelas aplicações que visualizam informação multimédia em tempo real. Os termos usados na língua inglesa para designar este tipo de transmissão é *live streaming* e as aplicações de vizualização chamam-se *stream players*. Estas aplicações permitem que um cliente de visualização apresente ao utilizador a informação que está a ser emitida por um emissor multimédia “no momento”.

As aplicações de tempo real dividem-se nas unidireccionais e nas interactivas, ver a Figura 9.6. No caso das primeiras, existe um só emissor e um ou mais receptores passivos da informação emitida. Os exemplos clássicos são os canais de televisão ou de rádio difundidos por redes de pacotes IP (IPTV ou *live streaming*) e os filmes ou a música difundidos pelo mesmo meio (*on demand streaming*). Os exemplos mais comuns de aplicações multimédia interactivas são as chamadas telefónicas sobre IP (VoIP), os jogos multimédia e as vídeo-chamadas, todas suportadas em redes de pacotes IP.

A informação multimédia transmitida e recebida em tempo real tem requisitos temporais e de débito relacionados com a resolução e a frequência de amostragem da informação sonora e visual, mas também tem requisitos temporais especiais devido à necessidade de sincronização dos interlocutores. Assim, estas aplicações têm exigências específicas de qualidade de serviço da rede, em particular no que diz respeito ao débito e tempo de trânsito extremo a extremo e ao *jitter*.

Débito extremo a extremo mínimo

O codec usado pelo emissor requer um dado débito mínimo extremo a extremo. Se a rede não o tiver disponível, é necessário “mudar de rede”, ou usar uma resolução

³ Os termos normalmente usados na língua inglesa para designar estas aplicações são *players* e *viewers* e as mesmas permitem a reprodução de som e a visualização de imagens com sonorização. Em português usaremos o termo “visualizador” como sinónimo de um programa ou dispositivo hardware / software que descodifica informação sonora ou visual e a reproduz, codificador àquele que apenas codifica, e emissor àquele que a transmite.

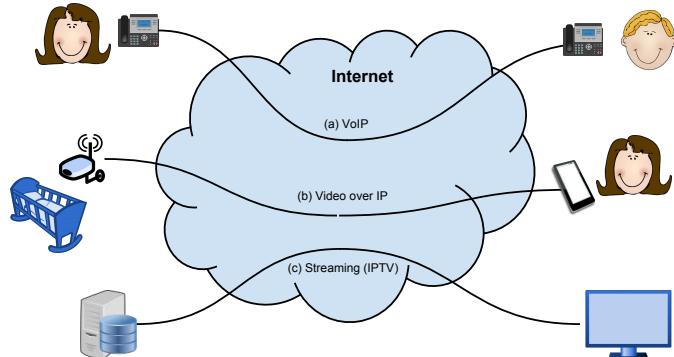


Figura 9.6: Tipos de aplicações multimédia em tempo real

mais baixa na informação multimédia transmitida. No limite, se o débito for variando com o tempo, a aplicação deveria adaptar-se dinamicamente à capacidade disponível e ajustar a resolução e a amostragem da informação multimédia transmitida.

Tempo de trânsito extremo a extremo máximo

Dado que a transmissão é em tempo real, a informação tem de ser visualizada até um limite de tempo máximo depois de ter sido produzida e emitida. Como todas as aplicações em tempo real ou ao vivo (*live*) têm de recorrer a um *playback buffer* para compensar o *jitter*, ver a Secção 3.4, ao tempo de trânsito extremo a extremo somam-se os atrasos com origem neste mecanismo de compensação das variações do tempo de trânsito, mas também os atrasos introduzidos pelas necessidades específicas do algoritmo de compressão e da transmissão em blocos de dados digitalizados característicos de cada codec. Assim, o atraso extremo a extremo visível pelo utilizador pode ser significativo.

O valor máximo do tempo de trânsito final aceitável é normalmente limitado, em particular numa aplicação interactiva, como uma aplicação de chamada telefónica (VoIP) podendo ser um pouco mais alargado em aplicações unidireccionais, como as aplicações de visualização de canais de televisão (IPTV), música ou de visualização de filmes.

No entanto, mesmo nesses casos, a paciência do utilizador tem limites pois este não está disposto a esperar vários minutos para começar a visionar um canal de televisão para o qual acabou de mudar. Mesmo com as aplicações não interactivas, o utilizador tolera no máximo alguns segundos até a informação começar (ou recomeçar) a ser visualizada e considera má qualidade de serviço esperar tempos superiores.

Jitter limitado

A rede, como sabemos, introduz *jitter*. Adicionalmente, os protocolos de transporte como o TCP podem acentuar significativamente o *jitter* extremo a extremo sempre que ocorrem erros, ou o ritmo de transmissão se altera devido aos mecanismos de controlo da saturação. Uma forma de combater estes dois factores é alongar o valor do *playout delay* a utilizar mas, como já referimos, esse alongamento tem limites.

Os estudos psicológicos demonstram que se um utilizador de uma aplicação interactiva de VoIP experimentar um intervalo de propagação utilizador a utilizador superior a 200 ms, os utilizadores finais começam a aperceber-se desse atraso, e o mesmo torna-se incomodativo a partir de 400 ms. O fenómeno é perceptível nas conversas observadas nos noticiários envolvendo reportagens com diálogo interactivo entre

o locutor local e um correspondente remoto noutro país distante através de um canal suportado em satélite, com atrasos claramente perceptíveis. O ritmo com que é possível fazer perguntas e obter respostas fica condicionado ao tempo de propagação extremo a extremo e um utilizador pouco experiente não consegue manter o diálogo.

Em conclusão, devido a fenómenos psicológicos dos utilizadores, as aplicações multimédia interactivas, para serem utilizadas de forma confortável, têm requisitos que implicam limites estreitos e rígidos ao *jitter*.

Todas as aplicações multimédia em tempo real têm requisitos de qualidade de serviço específicos que estão dependentes do tipo da aplicação, das expectativas dos utilizadores e do seu contexto de utilização. Esses requisitos dizem respeito ao débito, ao tempo de trânsito e ao *jitter*.

Os primeiros estão ligados às exigências dos codecs usados, os quais podem ser adaptados à capacidade disponível. Os requisitos mais difíceis de satisfazer são os ligados com o tempo de trânsito e o *jitter* e são particularmente importantes para as aplicações interactivas.

A questão que se coloca a seguir é saber que opções existem para transportar pela rede a informação multimédia destas aplicações.

Transmissão multimédia sobre TCP

O protocolo TCP é muito simples de utilizar e adapta-se à transmissão de todos os tipos de dados. Adicionalmente, devido a um fenómeno que será discutido na secção 10.4 e conhecido por “ossificação da Internet”, é o único protocolo de transporte que pode ser usado em certos contextos, pois os outros protocolos são bloqueados por equipamentos de segurança.

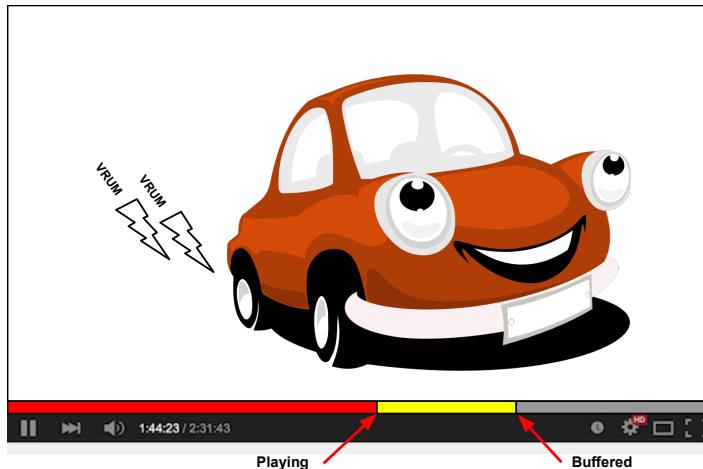


Figura 9.7: Utilização de um grande *playback buffer* em aplicações de *streaming* a pedido

Infelizmente, devido ao controlo de erros e ao controlo da saturação, o TCP tem um débito variável e, pior ainda, essas variações vêm acompanhadas de *jitter* suplementar, sobretudo em caso de erros. Para compensar esses problemas pode usar-se um *playback buffer* de grande dimensão, capaz de acomodar dezenas de segundos de visualização

mesmo sem chegarem novos dados, e um *playout delay* capaz de absorver variações pronunciadas da capacidade e do tempo de trânsito. É a solução mais comum nas aplicações de *streaming* de filmes ou de música como é ilustrado na Figura 9.7. No caso da música, como o débito necessário é mais baixo, o *playback buffer* pode ser mais pequeno.

Utilizar um *playback buffer* de grande dimensão conduziria a um compasso de espera significativo, quer no início da visualização, quer quando o utilizador resolve mudar a cena que pretende visualizar ou a música que pretende ouvir.

Mais recentemente começou-se a usar uma solução mais sofisticada, baseada em servidores contendo os filmes codificados em segmentos de alguns segundos de duração cada (*e.g.*, segmentos de 2 segundos), cada um dos quais disponível em diferentes resoluções. Compete ao visualizador fazer um pedido ao servidor a solicitar o envio de um segmento, indicando o seu índice (posição no filme) e a resolução escolhida.

Assim, o visualizador quando solicita o envio dos primeiros segundos do filme, pode optar por pedir segmentos codificados numa resolução mais baixa, e portanto requerendo um débito menor. Isso permite-lhe obter rapidamente segmentos do filme que pode ir mostrando ao utilizador. Se o utilizador mantiver uma visualização sequencial, o visualizador pode solicitar os segmentos seguintes com resolução mais elevada. Se a transição entre segmentos estiver alinhada com os *I-frames*, o utilizador acaba por não se aperceber das alterações de resolução.

Este mecanismo, para além de permitir um início de visualização mais rápido, permite também ao visualizador adaptar a resolução com que solicita os segmentos à capacidade da rede. Se os segmentos chegam mais depressa que a duração do filme neles codificado, o visualizador pode aumentar a resolução do próximo segmento a pedir, senão tem de diminuir a resolução dos segmentos seguintes. Trata-se de um controlo dinâmico e adaptativo da resolução, *i.e.*, da quantidade de informação pedida, à capacidade disponível na rede.

Esta solução torna-se ainda mais realista se os filmes forem replicados por diferentes servidores e os mecanismos que serão descritos no Capítulo 13 forem usados para dirigir os clientes para a réplica mais próxima. É a solução adoptada por serviços como o YouTube ou o Netflix, que usam conteúdos replicados em vários centros de dados espalhados por diferentes países. Como nesta situação os clientes estão relativamente mais próximos dos conteúdos a visualizar, é possível obter uma solução de qualidade razoável, baseada na transferência por TCP e na utilização de resolução variável. Uma solução semelhante também pode ser adoptada para a transmissão de canais de televisão em tempo real e para a transmissão de música. Neste último caso, na maioria das situações, as redes bem dimensionadas suportam a resolução mais alta.

É também possível utilizar TCP para transferir informação multimédia em aplicações interactivas. No entanto, neste caso, é habitual utilizar também resoluções variáveis mas é frequente o utilizador obter baixa qualidade final caso a capacidade da rede não seja suficiente, ou durante momentos de perda significativa de pacotes. É o exemplo típico de aplicações de vídeo-conferência interactivas gratuitas (*e.g.*, Skype, Google Hangouts, *etc.*).

O protocolo TCP é simples, omnipresente, compensa os erros e adapta-se à capacidade disponível na rede. Por estas razões constitui também um suporte adequado à transferência de informação multimédia se o débito médio extremo a extremo for o adequado ao codec usado.

Infelizmente, as suas virtudes têm como repercução a amplificação do *jitter* extremo a extremo, em particular na presença de erros e também devido à actuação dos mecanismos de controlo da saturação. Este defeito pode ser combatido usando um *playout delay* mais longo.

Apesar de essa solução ser compatível com aplicações unidireccionais, a mesma introduz atrasos suplementares e incómodos para os utilizadores finais, particularmente em aplicações multimédia interactivas, pelo que muitas vezes as aplicações recorrem à redução da resolução, para diminuírem os requisitos de capacidade de rede e minorarem os efeitos negativos sobre o *jitter*.

A introdução de mecanismos de adaptação dinâmica da resolução dos fluxos multimédia, à qualidade da rede, é uma solução comum que tenta minorar os efeitos do controlo de saturação do TCP. Quando as aplicações são interactivas, a necessidade de encurtar tanto quanto possível o *playout delay*, implica a redução da qualidade dos fluxos multimédia através do uso de codecs de pior qualidade.

No entanto, existem diversas situações que recomendam o uso do protocolo UDP. A primeira é quando se pretende enviar canais de televisão em simultâneo para uma grande quantidade de receptores e optimizar a utilização da capacidade da rede através da utilização de IP multicasting, ver a Secção 5.1. A segunda é quando se deseja satisfazer requisitos de interactividade estritos para suporte de chamadas telefónicas.

Transmissão multimédia sobre UDP

O TCP só suporta canais ponto-a-ponto. Quando se pretende servir muitos clientes simultaneamente, a solução TCP com um único servidor não escala. Nesse caso é possível recorrer a muitos servidores em simultâneo, solução só acessível caso exista um modelo de negócio viável e capacidade de investimento significativa. Também é possível recorrer a um modelo de distribuição do conteúdo através de uma solução P2P (ver a Secção 1.4), mas nesse caso é mais difícil manter um nível de qualidade adequado.

Quando os conteúdos são difundidos a pedido, estas são provavelmente as únicas soluções possíveis pois cada cliente visualiza um conteúdo diferente em cada momento. No entanto, quando se trata de canais de televisão, muitos utilizadores pretendem ver simultaneamente o mesmo conteúdo e uma solução de difusão de um para vários parece ser a mais adequada e eficiente. Esta solução pode ser realizada usando IP Multicasting, mas este só é acessível às aplicações recorrendo ao transporte UDP.

Para além de suportar *multicasting* ao nível transporte, o UDP tem a vantagem de não amplificar o *jitter* da rede, mas tem o defeito de não compensar os erros, pelo que quando se perdem datagramas UDP, o receptor vai receber informação multimédia com lacunas. Se a origem das perdas de datagramas UDP for só os erros nos canais, a sua taxa nos canais de fibra actuais é relativamente baixa, mas se a perda de pacotes se dever também a canais mal dimensionados, a taxa de erros é geralmente muito elevada, independentemente da qualidade dos canais.

Na secção seguinte serão analisadas várias técnicas de compensação dos erros em canais que transportam informação multimédia. No entanto, essas técnicas só se podem aplicar quando as perdas não são significativas nem sucedem em cascata. Por isso, não é muito realista usar UDP para transferir informação multimédia em redes mal dimensionadas ou com taxas de erro significativas. Por outro lado, devido aos modelos de negócio dominantes na Internet, a utilização de *multicasting* não é realista quando os pacotes difundidos têm de atravessar redes controladas por mais do que um operador. Assim, a utilização de UDP e de IP Multicasting para transferir a informação multimédia de canais de televisão está restringida ao contexto de um operador e respectivos clientes ligados directamente por canais de qualidade.

Esta solução é hoje em dia popular nos serviços de difusão de canais de televisão sobre IP (IPTV), em substituição das soluções tradicionais da televisão por cabo. O operador com serviço IPTV reserva capacidade no seu *backbone* para transmitir para

todos os comutadores regionais e urbanos fluxos de pacotes UDP por *multicasting* contendo todos os canais de televisão acessíveis aos clientes. Cada cliente, através de um canal rede dedicado, apenas subscreve os canais de televisão que pretende visualizar no momento e só recebe os fluxos de pacotes UDP correspondentes a esses canais.

Com canais rede de qualidade, uma rede bem dimensionada e reserva de capacidade para os fluxo de pacote UDP em *multicasting* correspondentes aos canais de televisão, a qualidade final é bastante boa e é possível usar um *playout delay* bastante curto. A utilização de *multicasting* favorece a escalabilidade do serviço, sendo possível servir centenas de milhar de clientes a partir de poucos emissores.

O facto de o protocolo UDP não dilatar o *jitter* é muito importante em aplicações interactivas e por isso o transporte de informação multimédia via UDP é também frequentemente usado nas aplicações de telefone e de vídeo conferência. Nestes casos também se utilizam técnicas de compensação dos erros tendo em consideração a natureza multimédia da informação trocada pelos interlocutores.

A transmissão de conteúdos multimédia sobre UDP tem a vantagem de ser compatível com a utilização de um *playout delay* curto, pois o protocolo não introduz dilatações suplementares do *jitter*. Adicionalmente, o protocolo é compatível com a difusão (*multicasting*), pelo que aumenta a escalabilidade do serviço quando existem muitos clientes a subscrever simultaneamente o mesmo conteúdo.

No entanto, como o UDP não compensa os erros da rede, o conteúdo pode apresentar lacunas. Quando a taxa de erros se deve apenas a erros nos canais e é pouco significativa, existem técnicas especiais de compensação dos erros de omisão de informação multimédia que são usadas para minorar o seu impacto sobre a qualidade final percepcionada pelos utilizadores.

9.3 Tratamento de erros em fluxos multimédia

Quando usamos um protocolo como o UDP para transferir informação multimédia, alguns datagramas UDP podem não chegar ao destino. Nesta secção vamos analisar algumas técnicas de compensação desses erros, algumas das quais exploram facetas específicas do tipo de informação transmitida. Com efeito, como introduzimos no início do capítulo, mesmo quando reduzimos a quantidade de informação multimédia transmitida, a utilidade da mesma para os utilizadores finais pode até continuar a ser idêntica (*e.g.*, numa conversa telefónica, apesar de ser mais difícil de perceber o som com erros, o destinatário pode continuar a perceber a frase do interlocutor).

A informação multimédia é sempre organizada em *frames*. Para ilustrar as técnicas usadas para compensar os erros vamos admitir, por hipótese, e sem perda de generalidade, que cada *frame* corresponde a um datagrama UDP, contido num pacote IP, e que o mesmo contém som digitalizado correspondente a um certo período de tempo.

Usando o codec G.711, que usa 8 bits para codificar cada amostra, e uma frequência de amostragem de 8 KHz, a cada milissegundo de som correspondem 8 bytes de informação. Para encher um *payload* com 1024 bytes de informação, seria necessário esperar 125 ms. Como a estes 125 ms é necessário juntar o tempo de propagação extremo a extremo do pacote de dados, e ainda um *playout delay* para acomodar o *jitter*, é fácil ultrapassar o limite a partir do qual os utilizadores se aperceberiam do atraso da transmissão numa conversa telefónica (atrasos > 150 ms). Por esta razão,

o período contido em cada pacote é mais pequeno. Com 20 ms cada pacote conterá apenas 160 bytes de som digitalizado.

Assim, durante a transmissão a perda de um pacote inutilizará 20 ms de som. Para evitar que o utilizador se aperceba desta falta, é necessário encontrar formas de a compensar.

FEC – Forward Error Correction

Como a informação é transmitida por pacotes e existem mecanismos de detecção de erros que rejeitam pacotes com erros, o tipo de erros que importa corrigir são designados erros de omissão. Assumindo que os pacotes têm um número de sequência, o receptor pode detectar qual o pacote em falta.

Existem códigos de correção de erros que consistem em informação redundante, acrescentada à informação transmitida, e que permite ao receptor detectar e corrigir eventuais erros que tiveram lugar durante a transmissão. Este tipo de técnicas chamam-se FEC (*Forward Error Correction*), por as mesmas se basearem em informação transmitida para a frente, sem recurso a informação retransmitida (voltando atrás). As mesmas podem também ser usadas a nível aplicacional para resolver o nosso problema pois existe uma classe de códigos de correção de erros de omissão, designados por *erasure codes*, que foram desenvolvidos para lidar com esses erros.

Uma das formas mais simples deste tipo de códigos consiste em transmitir periodicamente um pacote que contém um sumário, calculado pelo emissor, de um conjunto (de por exemplo 4) dos últimos pacotes transmitidos. Quando o receptor detecta a falta de um pacote, pode usar o sumário para calcular o valor do pacote em falta. Por exemplo, seja $s = x_1 + x_2 + x_3 + x_4$ o sumário. Caso falte o pacote x_2 , o seu valor pode ser assim calculado: $x_2 = s - (x_1 + x_3 + x_4)$.

Para se perceber uma forma concreta de aplicar a técnica na prática sem usar aritmética de números representados em centenas ou milhares de bits, vamos assumir que cada pacote consiste num único bit. O tipo de sumário mais simples consiste num sumário correspondente ao resultado do XOR (*eXclusive OR*, operação a seguir denotada por \oplus) do conteúdo dos últimos 4 pacotes, *i.e.*, dos últimos 4 bits, transmitidos. Seja o sumário $s = x_1 \oplus x_2 \oplus x_3 \oplus x_4$, caso falte o bit x_2 , então $x_2 = s \oplus x_1 \oplus x_3 \oplus x_4$ visto que com a operação *eXclusive OR*, a soma e a subtração são equivalentes. Por exemplo, se os últimos 4 bits transmitidos foram 1, 1, 0, 0, o seu sumário é 0. Se faltar o segundo bit, é fácil deduzir que o seu valor era 1, pois $x_2 = 0 \oplus 1 \oplus 0 \oplus 0$. Dado que a operação realizada é equivalente ao cálculo da paridade ímpar, estes códigos também se chamam códigos de paridade.

Um pacote não contém um bit mas sim uma grande sequência deles. O raciocínio feito para um bit é agora feito para cada um dos bits de cada pacote como se estes estivessem alinhados coluna a coluna como mostra a Figura 9.8.

Assim, o receptor vai recebendo os pacotes, mas sempre que lhe falta um, espera pelo próximo pacote de sumário e calcula o valor do que está em falta, ver a figura 9.9. É claro que se o pacote em falta for apenas o do sumário não há problema, mas se entre dois sumários faltar mais do que um pacote, não é possível reconstituir os pacotes faltantes.

Esta técnica tem o defeito de introduzir uma percentagem significativa de bits redundantes. Por exemplo, com sumários de 4 pacotes introduz 25% de redundância. Para além disso, em caso de erro, multiplica por 4 o tempo necessário para processar cada pacote. Tal pode não ser muito grave caso o *playout delay* acomode o tempo necessário para encontrar o pacote com o sumário a tempo. Para diminuir a probabilidade de faltar mais do que um pacote entre cada sumário, é possível reduzir cada sumário ao sumário de três, dois ou mesmo de um único pacote. Neste último caso cada pacote é idêntico ao seu sumário, mas necessita-se do dobro do débito requerido pelo codec. No caso do G.711, ao invés de 64 Kbps, seriam necessários 128 Kbps.

x_1	0 1 0 0 1 1 0 ... 0 1 1 0 0 1
x_2	1 1 1 0 1 1 0 ... 0 1 0 0 0 1
x_3	0 0 0 1 1 0 0 ... 0 1 1 1 1 1
x_4	0 1 0 0 1 0 0 ... 0 1 1 0 0 1
s	1 1 1 1 0 0 0 ... 0 0 1 1 1 0
$x_1 + x_3 + x_4$	0 0 0 1 1 1 0 ... 0 1 1 1 1 1
$+s = x_2$	1 1 1 0 1 1 0 ... 0 1 0 0 0 1

Figura 9.8: Cálculo do pacote sumário e sua utilização para calcular o valor de um pacote em falta

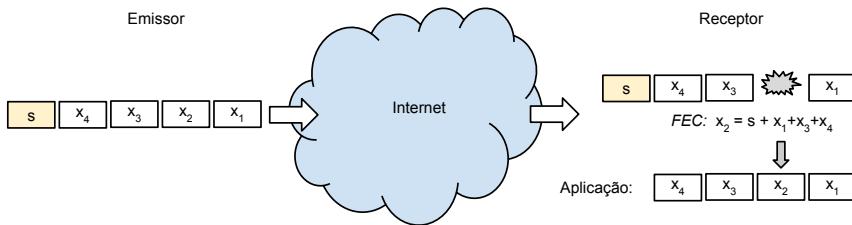


Figura 9.9: Correcção de erros de omissão usando *Forward Error Correction* com base em informação redundante em sumários

A técnica que acabámos de ilustrar é particularmente útil caso se esteja a utilizar IP Multicasting, pois as técnicas de retransmissão de pacotes em falta não são aplicáveis à difusão de um para muitos. De facto, as perdas de pacotes neste cenário são de diferentes pacotes pois cada receptor é servido por diferentes canais. No entanto, o mecanismo permite recuperar de diferentes erros de omissão, ver a Figura 9.10.

Existem outras técnicas de FEC que não introduzem informação redundante mas que se baseiam em tentar reconstruir a informação omitida a partir daquela que a antecede ou daquela que a segue. Os pacotes com 20 ms de amostragem de som contém 160 amostras. Reconstruir os 160 valores em falta é mais difícil, mas é possível usar uma técnica que torna a tarefa mais simples.

Emparelhamento (*Interleaving*)

Se tomarmos os 160 valores de um pacote e os 160 valores do pacote seguinte, é possível enviar no primeiro pacote os bytes pares de ambos os pacotes e no seguinte os bytes ímpares, ver a Figura 9.11. Se um dos pacotes se perder, o receptor, apesar de tudo, dispõe da informação dos dois pacotes mas com uma frequência de amostragem de metade da verdadeira e poderá, por interpolação, obter os valores perdidos.

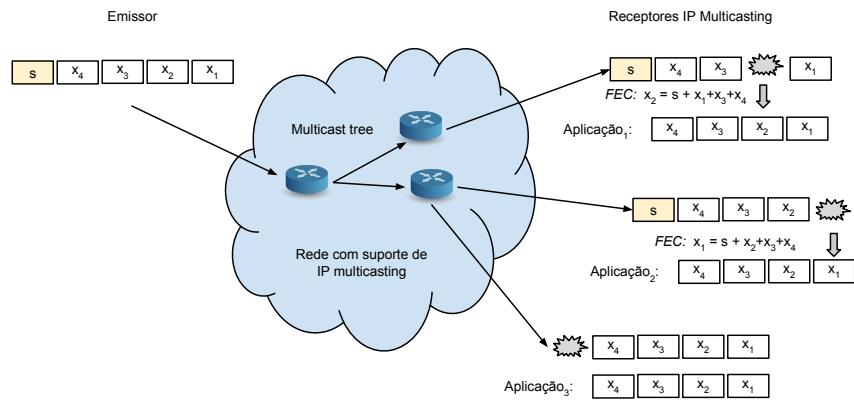


Figura 9.10: Correcção de erros de omissão com *multicasting* usando *Forward Error Correction*

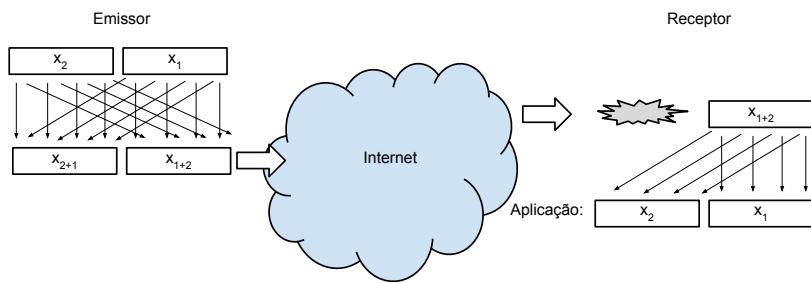


Figura 9.11: *Forward Error Correction* com *interleaving* byte a byte

A técnica ilustrada tem o mérito de não requerer o aumento do débito pois não introduz informação redundante. A mesma baseia-se em obter os bytes em falta por interpolação a partir de amostras com pior amostragem dos valores originais. No entanto, a mesma duplica o tempo necessário para processar cada pacote perdido pois o mesmo só pode ser reconstruído quando chega o seguinte.

Adicionalmente, é impossível usá-la com *frames* que contenham informação multimédia comprimida pois não é possível realizar interpolação sobre a mesma. No entanto, é possível aplicá-la usando outra alternativa que consiste em decompor cada pacote num número limitado de sub-intervalos independentes, por exemplo 4, e transmitir cada sub-intervalo em pacotes distintos, ver a Figura 9.12.

Utilizando 4 sub-intervalos de 5 ms cada, caso um pacote seja omitido, é possível reconstruir cada um de 4 pacotes com 5 ms de som em falta em cada um, o que diminui o impacto da falha para apenas 40 bytes. Fazendo uma decomposição num número superior de pacotes diminuiria ainda mais o impacto das falhas, mas aumenta a pressão sobre o *playout delay* na medida em que só é possível recompor cada pacote depois de terem chegado todos os seus sub-intervalos.

Para terminar esta secção vamos ainda ilustrar outra técnica de compensação de erros que consiste em enviar simultaneamente dois fluxos multimédia, com resoluções distintas, e deslocados um do outro.

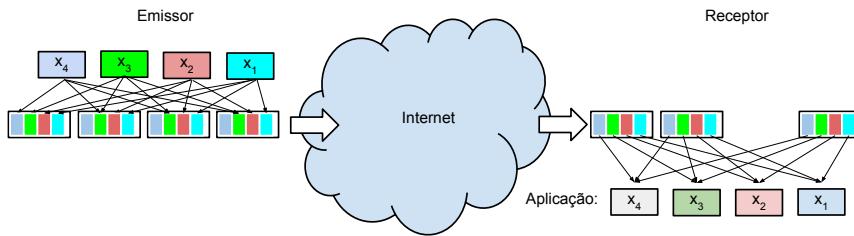


Figura 9.12: *FEC – Forward Error Correction* com *Interleaving* de sub-intervalos

Fluxos emparelhados

Supondo que o fluxo base é o correspondente ao codec G.711 e o fluxo de menor resolução é o correspondente ao codec G.709, que requer apenas cerca de 1/8 do débito do G.711, cada 20 ms de som ocupa 160 bytes no primeiro e cerca de 20 bytes no segundo. Assim, cada pacote contém o *frame* correspondente ao respectivo fluxo G.711, assim como o *frame* correspondente ao pacote anterior codificado segundo o codec G.729. No total, ao invés de 160 bytes, cada pacote tem 180 bytes de informação.

Se um pacote se perder, o pacote seguinte contém o som correspondente, só que com uma resolução inferior. Tal permite ao *player* recompor o som em falta apesar de com pior qualidade. A Figura 9.13 ilustra a técnica.

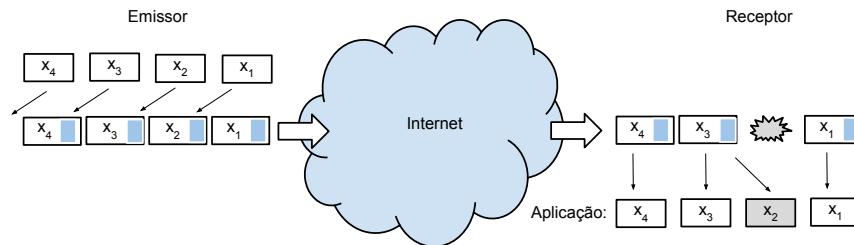


Figura 9.13: *Forward Error Correction* com base num codec de resolução inferior

Esta técnica permite recuperar a perda de um pacote. É possível generalizá-la para mascarar a perda de n pacotes incluindo em cada pacote uma versão comprimida dos n pacotes anteriores. Assim é possível mascarar perdas de maior dimensão à custa de mais fluxos redundantes, *i.e.*, maior débito afectado ao fluxo multimédia, e aumentando o tempo necessário para processar a informação em caso de perdas. Novamente, caso o *playback buffer* e o *playout delay* acomodem este tempo extra, não haverá problemas.

Quando se utiliza um protocolo como o UDP para transporte de informação multimédia, podem surgir erros devido à omissão de pacotes. Para mascarar o seu efeito é possível usar técnicas gerais de FEC (*Forward Error Correction*) com base em códigos de correção de erros do tipo *erasure*.

No entanto, explorando as propriedades da informação multimédia, é também possível reconstruir a informação em falta usando interpolação de forma tal que a mesma

seja realista e o utilizador não se aperceba da perda de resolução. No limite, é sempre possível reutilizar o último *frame* recebido para cancelar uma perda, particularmente quando se trata de imagens.

Finalmente, para terminarmos esta panorâmica do transporte de informação multimédia, na secção seguinte é introduzido o protocolo RTP – *Real-time Transport Protocol*, que foi definido para suportar a transmissão de informação multimédia em tempo real.

9.4 RTP – *Real-time Transport Protocol*

Para que uma aplicação multimédia funcione, é necessário que as várias partes em comunicação se entendam sobre o número, o formato e o sincronismo dos fluxos multimédia que circulam pela rede. Quando a aplicação e o software envolvido são proprietários (*e.g.*, como no caso da aplicação Skype) não é preciso nenhum investimento suplementar em normalização. No entanto, quando se pretende atingir uma situação em que várias aplicações, desenvolvidas por equipas ou fabricantes distintos, possam funcionar em conjunto, é necessário tentar normalizar os formatos e os protocolos usados para garantir essa inter-operação.

As aplicações multimédia que usam redes são predominantemente aplicações em tempo real, uni ou multi-direcionais, em que os problemas do tempo de trânsito extremo a extremo são críticos, e por isso investiu-se na normalização do transporte de segmentos de informação multimédia (e não de fluxos contínuos sem delimitações de mensagens como o TCP).

A abordagem seguida foi definir um protocolo que completasse o protocolo UDP com funcionalidades comuns à maioria das aplicações multimédia do tipo *live streaming* (*e.g.*, IPTV), ou para suporte de conversas telefónicas e video-conferências (*e.g.*, VoIP e *Conference Call over IP*). Dessa forma conseguiam-se dois objectivos: aplicações desenvolvidas de forma independente poderiam inter-operar, e poupava-se o trabalho de reinventar a roda em cada aplicação.

Apesar da motivação original, nada impede os segmentos RTP de serem transportados sobre TCP, apesar de nesse caso existirem informações inúteis no cabeçalho e faltarem algumas como por exemplo a dimensão de cada segmento.

O resultado final foi um protocolo designado RTP (*Real-time Transport Protocol*). Como mostra a Figura 9.14, o RTP é uma especialização (no sentido da derivação de classes na orientação por objectos) do UDP para transporte de *frames* multimédia pertencentes a um ou mais fluxos multimédia. A Figura 9.14 mostra a inserção do RTP na pilha de protocolos TCP/IP. As mensagens do RTP contêm elementos do nível aplicação e por esse motivo deveríamos chamar-lhes mensagens ou *frames* multimédia e não segmentos ou pacotes. No entanto, a tradição manda que se use o termo pacote RTP, que usaremos também.

A Figura 9.15 mostra o cabeçalho do protocolo RTP. Os primeiros 12 bytes do cabeçalho, *i.e.*, as primeiras três linhas do cabeçalho RTP, são obrigatórias, enquanto que os campos CSRC e de extensão são opcionais, e são específicos daquilo que a norma designa como perfis (*profiles*), que são específicos de categorias de aplicações ou de tipos de informação multimédia. Procurou-se que o cabeçalho se restringisse aos campos comuns a todos os perfis, deixando para cabeçalhos opcionais suplementares o suporte de funcionalidades específicas. Os campos comuns são os mais importantes para se perceber a filosofia do protocolo e são os únicos a seguir explicados.

O cabeçalho tem inicialmente um conjunto de campos de pequena dimensão, quase todos campos do tipo *flags*, que servem para indicar a versão do protocolo, se o *payload* contém ou não informação de *padding*, se está presente ou não um cabeçalho de extensão, *etc.* Um desses campos, o campo CC, será referido a seguir quando for explicado o papel do campo CSRC.

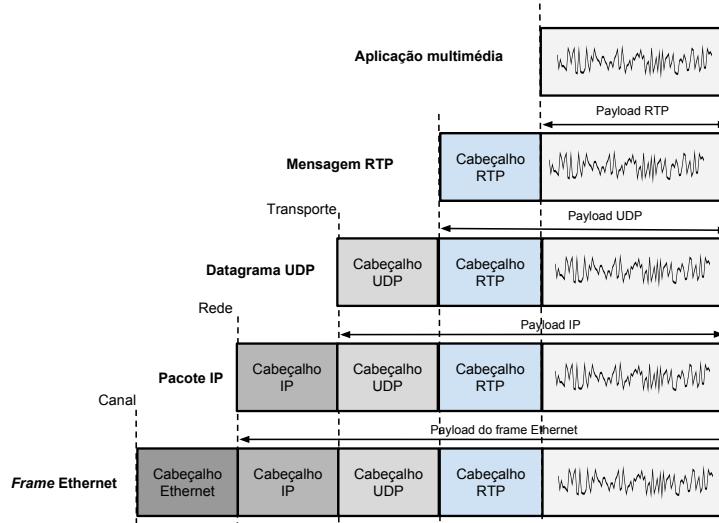


Figura 9.14: Inserção do protocolo RTP na pilha de protocolos TCP/IP

O campo **Payload Type** tem 7 bits e permite codificar o formato do fluxo, nomeadamente o codec usado. A tabela dos codecs possíveis depende do perfil usado pela aplicação e em função do mesmo estão definidas tabelas de codecs. O RFC 3551 define o perfil usado para conferências multimédia. A Tabela 9.1 lista um subconjunto dos valores de **Payload Type** definidos no mesmo. A indicação do tipo de codificação em cada pacote RTP permite alterar a qualquer momento o tipo usado pelo emissor sem necessidade de negociar o mesmo com o ou os receptores.

O campo **Sequence number** permite numerar os pacotes para que o receptor possa detectar as falhas e colocar os pacotes na ordem de emissão. Nenhum mecanismo de ACK e retransmissão está previsto no protocolo. Ao invés disso, os esquemas de FEC ou similares, anteriormente descritos, são usados para lidar com as falhas.

O campo **Timestamp** tem 32 bits e caracteriza a posição no fluxo do início do *payload* do pacote. Este valor não representa um tempo absoluto, nem a norma do RTP define as unidades em que se exprime. O valor inicial é fixado pelo emissor quando inicia o fluxo, e a unidade de progresso é definida pelo tipo do fluxo. Por exemplo, se o codec for o G.711, e um pacote contiver 20 ms de som, isso quer dizer que o valor do campo **Timestamp** do pacote seguinte deverá ser normalmente incrementado de 160 face ao deste pacote. O mecanismo de **Timestamp** é também usado para sincronizar a visualização de vários fluxos, mesmo que os seus valores iniciais e as suas escalas sejam específicas a cada um deles. A indicação de uma **Timestamp** em cada pacote permite substituir o silêncio num fluxo sonoro pela ausência de envio de *frames*.

O campo **Synchronisation source identifier (SSRC)** também tem 32 bits e é um número aleatório que identifica univocamente a origem de um fluxo. Por exemplo, numa conferência multimédia, cada participante usa um número destes diferentes. Desta forma a aplicação não está dependente dos endereços IP ou das portas para distinguir as diferentes fontes de informação multimédia existentes. Por exemplo, o mesmo computador, quer tenha um só endereço IP ou vários diferentes e a mudarem dinamicamente, pode ser a origem de uma ou mais fontes de informação multimédia.

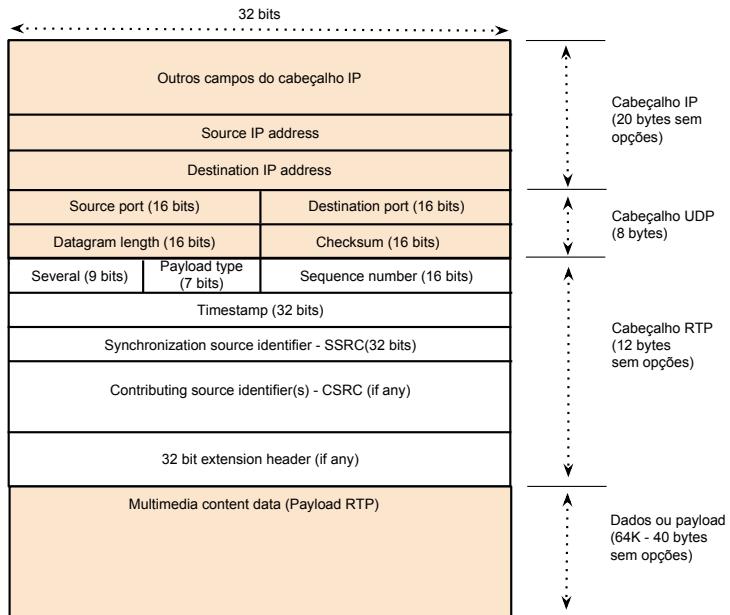


Figura 9.15: Cabeçalho do protocolos RTP

cuja identificação é independente de endereços IP e portas.

Existe um caso particular em que o SSRC não representa um utilizador, mas um misturador, que junta num só fluxo sonoro o som de vários participantes. Neste caso o misturador tem um identificador próprio, mas o campo *Contributing source identifiers (CSRC)* contém a lista das fontes misturadas. A flag CC é posicionada para assinalar que o pacote foi criado por um misturador.

O protocolo RTP é definido pelo RFC 3550. Associado ao mesmo existe outro protocolo, designado RTCP (*Real-time Transport Control Protocol*) que permite enviar periodicamente informação de controlo associada a um fluxo RTP (e.g., *feedback* sobre a qualidade da recepção, formas de sincronizar diferentes fluxos vindos do mesmo utilizador, etc.).

Para fomentar a inter-operação entre aplicações multimédia em tempo real que enviem dados multimédia encapsulados em UDP, foram normalizados os protocolos RTP (*Real-time Transport Protocol*) e RTCP (*Real-time Transport Control Protocol*).

O primeiro destes protocolos normaliza um cabeçalho contendo um conjunto de informações úteis sobre os dados multimédia enviados em cada pacote, entre os quais a identificação dos emissores da informação, o tipo de codificação usado (codec), um número de sequência que permite detectar perdas e uma indicação temporal para permitir a sincronização dos pacotes do mesmo ou de vários fluxos. O segundo normaliza o envio de informações sobre a qualidade da recepção, formas de sincronizar diferentes fluxos vindos do mesmo utilizador, etc.

Tabela 9.1: Codificação de alguns valores do campo do Payload Type segundo o RFC 3551 (lista parcial)

Código	Nome	Tipo	F. amostragem (Hz)	Débito
0	PCMU (G711)	Audio	8.000	64 Kbps
3	GSM	Audio	8.000	13 Kbps
4	G723	Audio	8.000	
6	DVI4	Audio	16.000	
7	LPC	Audio	8.000	2,4 Kbps
11	L16	Audio	44.100	
18	G729	Audio	8.000	
14	MPEG Audio	Audio	90.000	
26	Motion JPEG	Video	90.000	
31	H.261	Video	90.000	
32	MPEG 1	Video	90.000	
33	MPEG 2	Video	90.000	

9.5 Resumo e referências

Resumo

O som digitalizado consiste numa sequência de valores numéricos que representam a intensidade da onda sonora medida a intervalos regulares. A resolução da digitalização corresponde ao número de bits usados para representar os valores medidos. A frequência de amostragem determina as frequências presentes na representação digital. É normal usar 8 bits de resolução e 8 KHz de frequência de amostragem para a digitalização da voz nos sistemas telefónicos, ou 16 bits / 44,1 KHz para a música, ou ainda maior resolução e frequência de amostragem quando existem requisitos de muito alta fidelidade.

Devido às características percepcionais do sentido da audição é possível aplicar algoritmos de compressão do sinal sonoro digitalizado. Estes algoritmos podem implicar perda de informação (são *lossy*) mas não perda de significado nos sistemas telefónicos, nem diferenças demasiado perceptíveis da qualidade musical. Os ganhos assim obtidos podem reduzir o débito do fluxo digital correspondente ao som digitalizado até uma ordem de grandeza, sem um impacto significativo na utilidade final da informação sonora transmitida.

Uma imagem digitalizada contém uma sequência de valores que codificam a cor e a intensidade de um conjunto de pixels, *i.e.*, de unidades elementares da imagem. O número de bits usados para codificar o valor de cada pixel e a densidade de pixels da imagem condicionam a sua qualidade. Um filme é uma sequência de imagens justapostas (*image frames*), com frequências entre 20 e 60 imagens por segundo.

O débito de informação por unidade de tempo requerido para a transmissão de imagens fixas e de filmes é muito elevado se não se utilizarem algoritmos de compressão. Dadas as características de redundância das imagens e dos filmes, assim como as características da vista humana, é possível utilizar algoritmos de compressão de imagem e de filmes que reduzem esse débito até 3 ordens de grandeza, através de perda de informação, mas muitas vezes com resultados imperceptíveis para a vista.

A informação multimédia tem características particulares que permitem que a “mesma” informação seja transmitida com resoluções distintas, ou mesmo com erros, sem que a sua utilidade seja afectada na mesma proporção.

Esta característica, a par de outros requisitos das aplicações que usam informa-

ção multimédia, tem um impacto significativo sobre as técnicas de transmissão de informação usadas sobre uma rede de pacotes.

Todas as aplicações multimédia em tempo real têm requisitos de qualidade de serviço específicos que estão dependentes do tipo da aplicação, das expectativas dos utilizadores e do seu contexto de utilização. Esses requisitos dizem respeito ao débito, ao tempo de trânsito e ao *jitter*.

Os primeiros estão ligados às exigências dos codecs usados, os quais podem ser adaptados à capacidade disponível. Os requisitos mais difíceis de satisfazer são os ligados ao tempo de trânsito e *jitter*, e são particularmente importantes para as aplicações interactivas.

O protocolo TCP é simples, omnipresente, compensa os erros e adapta-se à capacidade disponível na rede. Por estas razões constitui também um suporte adequado à transferência de informação multimédia se o débito médio extremo a extremo for o adequado ao codec usado.

Infelizmente, as suas virtudes têm como repercussão a amplificação do *jitter* extremo a extremo, em particular na presença de erros e também devido à actuação dos mecanismos de controlo da saturação. Este defeito pode ser combatido usando um *playout delay* mais longo.

Apesar de essa solução ser compatível com aplicações unidireccionais, a mesma introduz atrasos suplementares e incómodos para os utilizadores finais, pelo que muitas vezes as aplicações recorrem à redução da resolução para diminuírem os requisitos de capacidade de rede e minorarem os efeitos negativos sobre o *jitter*.

A introdução de mecanismos de adaptação dinâmica da resolução dos fluxos multimédia à qualidade da rede é uma solução comum que tenta minorar os efeitos do controlo de saturação do TCP. Quando as aplicações são interactivas, a necessidade de encurtar tanto quanto possível o *playout delay*, implica a redução da qualidade dos fluxos multimédia através do uso de codecs de pior qualidade ou com parâmetros ajustados.

A transmissão de conteúdos multimédia sobre UDP tem a vantagem de ser compatível com a utilização de um *playout delay* curto, pois o protocolo não introduz dilatações suplementares do *jitter*. Adicionalmente, o protocolo é compatível com a difusão (*multicasting*), pelo que aumenta a escalabilidade do serviço quando existem muitos clientes a subscrever simultaneamente o mesmo conteúdo.

No entanto, quando se utiliza um protocolo como o UDP para transporte de informação multimédia, podem surgir erros devido à omissão de pacotes. Para mascarar o seu efeito é possível usar técnicas gerais de FEC (*Forward Error Correction*) com base em códigos de correcção de erros do tipo *erasure*.

No entanto, explorando as propriedades da informação multimédia, é também possível reconstruir a informação em falta usando interpolação de forma tal que a mesma seja realista e o utilizador não se aperceba da perda de resolução. No limite, é sempre possível reutilizar o último *frame* recebido para cancelar uma perda, particularmente quando se trata de imagens.

Para fomentar a inter-operação entre aplicações multimédia em tempo real que enviem dados multimédia encapsulados em UDP, foram normalizados os protocolos RTP – *Real-time Transport Protocol* e RTCP – *Real-time Transport Control Protocol*.

O primeiro destes protocolos normaliza um cabeçalho contendo um conjunto de informações úteis sobre os dados multimédia enviados em cada pacote, entre os quais a identificação dos emissores da informação, o tipo de codificação usado (codec), um número de sequência que permite detectar perdas e uma indicação temporal para permitir a sincronização dos pacotes do mesmo e de vários fluxos.

Alguns dos termos introduzidos no capítulo são a seguir passados em revista. Entre parêntesis figura a tradução mais comum em língua inglesa.

Conversor analógico - digital (*ADC – Analog to digital converter*) Dispositivo que converte uma onda eléctrica analógica numa sequência de valores numéricos.

Conversor digital - analógico (*DAC - Digital to Analog Converter*) Dispositivo que converte uma sequência de valores numéricos numa onda elétrica analógica.

Frequência de amostragem (*Sampling rate*) Frequência com que um conversor analógico - digital produz amostras do sinal analógico.

Resolução das amostras (*Sample quantization*) Número de bits com que o valor de cada amostra é expresso por um conversor analógico - digital.

Compressão (*Compression*) Os fluxos multimédia contêm uma grande quantidade de informação e por isso os mesmos são comprimidos para serem transmitidos. As técnicas de compressão usadas podem ser sem perda de informação (*lossless*) ou com perda de informação (*lossy*).

Fidelidade (*Fidelity*) A informação multimédia depois de digitalizada e comprimida, é memorizada ou transmitida e depois descomprimida e finalmente restituída aos utilizadores. A fidelidade é uma propriedade subjectiva e qualitativa que capta a maior ou menor percepção pelos utilizadores finais da equivalência entre o sinal original e o restituído. Os utilizadores podem considerar que existe alta fidelidade mesmo que no processo se tenha perdido informação devido às características dos sentidos auditivo e visual. As técnicas usadas dizem-se psico-acústicas e psico-visuais.

Pixel (*Pixel*) Unidade de informação elementar e não decomponível sobre a cor e a intensidade de uma imagem. Uma imagem é formada por uma grande quantidade de pixels e a sua fidelidade exprime-se em termos da densidade dos pixels por unidade de superfície, assim como da quantidade de bits usados para exprimir a cor e a transparência.

RGB (*Red Green Blue*) Forma de codificação de qualquer cor como uma composição das cores vermelho, verde e azul em diferentes proporções.

Frame rate Um filme é composto por uma sequência de imagens cuja quantidade por unidade de tempo condiciona a sua qualidade. A vista humana é incapaz de distinguir as diferentes imagens desde que o seu número por unidade de tempo seja superior a 20 ou 30 por segundo. Para garantir conforto da visualização dos filmes é frequente usar 50 imagens por segundo. Essas imagens também se podem chamar fotogramas.

Codec (*CODer / DECoder*) Dispositivo hardware, software ou misto, capaz de codificar e descodificar informação multimédia codificada de acordo com um formato específico. Muitas vezes, informalmente, o termo é usado para designar quer o dispositivo, quer o formato.

Qualidade de serviço em multimédia (*Multimedia QoS*) A transmissão de um fluxo multimédia em tempo real (com “simultaneidade” entre a sua transmissão e consumo) requer da rede um conjunto de garantias de qualidade de serviço, nomeadamente débito mínimo e tempo de trânsito e *jitter* limitados. As exigências temporais são particularmente importantes e as mais difíceis de garantir.

Correcção de erros sem retransmissão (*FEC – Forward Error Correction*) Técnicas usadas para mascarar os erros que se produzem na transmissão de dados, sem retransmissão dos dados chegados com erros, baseada na transmissão de informação suplementar que permite compensar a informação errada.

Erros de omissão (*Erasure errors*) Erros caracterizados por a informação recebida conter omissões face à informação transmitida, mas a informação que é de facto recebida não contém erros. Os códigos FEC especializados em compensação de erros de omissão dizem-se *erasure codes*.

Correcção de erros por emparelhamento (*Interleaving FEC*) Técnica de correcção de erros baseada em decompor cada unidade de informação a transmitir em

subconjuntos e na transmissão de pacotes contendo um emparelhamento de subconjuntos de diferentes unidades. Caso se perdam pacotes, esta técnica permite compensar os erros de omissão por interpolação sobre a informação parcial recebida.

RTP – Real-time Transport Protocol Protocolo que enriquece a noção de *frame* multimédia com a transmissão de informações particularmente adequadas ao transporte de fluxos multimédia, nomeadamente: tipo do codec dos dados transmitidos na parte de dados, números de sequência para detecção de erros de omissão, informação de sincronização temporal, *etc.* O protocolo é complementado com o protocolo RTCP – *Real-time Transport Control Protocol* que permite enviar periodicamente informação de controlo associada a um fluxo RTP (*e.g.*, *feedback* sobre a qualidade da recepção, formas de sincronizar diferentes fluxos vindos do mesmo utilizador, *etc.*).

Referências

Este capítulo apresenta uma breve introdução aos conceitos base sobre a informação multimédia. O leitor interessado em aprofundar o tema poderá consultar um dos livros recomendados pelo SIGMM (*Special Interest Group on MultiMedia*) da ACM (ver mais abaixo a referência para a página Web do SIGMM).

As técnicas de correcção de erros de omissão independentes do tipo de informação transmitida são uma área muito activa, com aplicações à difusão de informação multimédia, aos sistemas móveis e aos sistemas de arquivo de informação. O leitor interessado poderá consultar os RFCs 5053 e 5510 assim como os artigos [Rizzo, 1997; MacKay, 2005]. O artigo [Perkins et al., 1998] apresenta uma síntese das técnicas de recuperação de erros de omissão usando as propriedades da informação multimédia.

O protocolo RTP é um protocolo com a ambição de suportar uma família de aplicações muito variada e cujos requisitos estão em constante evolução. O argumento “end-to-end” recomenda um grande controlo das aplicações sobre as funcionalidades disponibilizados pelas camadas inferiores, para evitar duplicação de funcionalidades. O sucesso do arquitecto de protocolos está intimamente ligado à sua capacidade de definir o que fica em cada nível. O artigo [Clark and Tennenhouse, 1990] propõe formas de navegar neste labirinto com especial ênfase nas aplicações multimédia.

Apontadores para informação na Web

- <http://ietf.org/rfc.html> – É o repositório oficial dos RFCs, nomeadamente dos citados neste capítulo.
- <http://sigmm.org/Education> – Página oficial do SIGMM com recursos educacionais como cursos, vídeos educacionais, livros e software.
- <http://www.itu.int/rec/T-REC-G.114-200305-I/en> – Dá acesso à recomendação G.114 da ITU “One-way Transmission Time” de 2003, que discute o papel do tempo de trânsito extremo a extremo no telefone.
- <http://www.jpeg.org> – Site oficial dos membros do comité que desenvolve a norma JPEG.
- <https://en.wikipedia.org/wiki/JPEG> – Entrada na Wikipedia sobre as normas JPEG e contém uma interessante descrição dos aspectos técnicos do conjunto das normas JPEG.
- <http://mpeg.chiariglione.org> – Site oficial dos membros do comité que desenvolve a norma MPEG.

9.6 Questões para revisão e estudo

1. Numa empresa existe um serviço de atendimento aos clientes via telefone, suportado em VoIP, em que todas as chamadas telefónicas são gravadas. A chamada média dura 320 segundos. O codec usado é o G.711 com frequência de amostragem de 8 KHz e um byte de resolução por amostra, produzindo um débito de 64 Kbps. Supondo que se pretende gravar as chamadas, quanto espaço em disco é necessário para gravar, sem compressão, uma chamada de duração média nas seguintes situações.
 - (a) A gravação contém todo o som digitalizado dos dois interlocutores sem mistura. A gravação inclui todos os períodos de silêncio, em que nenhum dos interlocutores fala.
 - (b) A gravação contém todos os pacotes RTP trocados entre os dois interlocutores admitindo que o codec transmite em cada pacote 20 ms de som. A gravação inclui todos os períodos de silêncio, em que nenhum dos interlocutores fala, e todos os cabeçalhos (IP, UDP e RTP).
 - (c) Questão idêntica à alínea anterior mas a gravação contém pacotes RTP contendo todos os sons correspondentes à conversa mas misturados num único fluxo sonoro.
 - (d) Sugira, continuando a usar o formato RTP, formas de diminuir a quantidade de informação guardada continuando a usar o mesmo formato para o som.
2. Pretende-se gravar em disco um CD de música com a duração de uma hora, sem compressão. Quanto espaço em disco é necessário?
3. Uma chamada telefónica é suportada no protocolo RTP e o codec usado é o G.711 com frequência de amostragem de 8 KHz e um byte de resolução por amostra, produzindo um débito de 64 Kbps. Os pacotes contêm 20 ms de som e têm a dimensão 160 bytes mais cabeçalhos. O tempo de trânsito entre o emissor e o receptor varia entre 50 e 200 ms. O receptor usa um *playout delay* fixo de 150 ms. Na tabela abaixo estão indicados os momentos de chegada dos pacotes ao receptor.
 - (a) Complete a tabela indicando o momento em que cada pacote começa a ser reproduzido.
 - (b) Qual a dimensão do *playback buffer* necessário para acomodar o *playout delay* no receptor.
4. Pretende-se gravar em disco um filme com a duração de uma hora, sem compressão. O filme está gravado usando 50 *frames* por segundo, cada *frame* tem 600×400 pixels a cores, e cada pixel é codificado em 2 bytes. Quanto espaço em disco é necessário?
5. Um servidor de *on-demand streaming* envia um fluxo contínuo com o débito constante de 2 Mbps de informação para um cliente por TCP. O tempo de trânsito entre o servidor e o cliente é de 120 ms. O cliente tem um *playback buffer* com 4 Mbytes que momentaneamente está cheio. Admitindo que o TCP do emissor pára de enviar dados devido a um episódio de saturação momentâneo, quanto tempo máximo pode durar essa paragem? Despreze os factores de segurança necessários para acomodar as necessidades de informação anterior ou posterior para a restituição do som e da imagem.
6. Um *frame* MPEG tem um erro, quantos *frames* de um filme podem ser afectados por esse erro: um ou vários? Justifique a sua resposta.

Tabela 9.2: Pacotes e momento da sua chegada

Número de seq. do pacote	Momento da chegada (ms)	Momento para ser “tocado”
1	85	
2	125	
4	145	
5	160	
3	180	
8	220	
6	245	
7	285	
9	290	
10	310	

7. Verdade ou mentira? Os códigos de correcção de erros do tipo *erasure* conseguem por si sós corrigir erros com origem no ruído nos canais que alteram alguns bits nos pacotes. Justifique a sua resposta.
8. Verdade ou mentira? O código de correcção de erros do tipo FEC conhecido por código de paridade podia ser aplicado à transmissão de ficheiros, evitando toda a retransmissão de pacotes. Justifique a sua resposta.
9. Qual a percentagem de informação de controlo de erros suplementar introduzida por um código de correcção de erros do tipo FEC usando paridades quando cada sumário cobre os 3 pacotes anteriores? Justifique a sua resposta.
10. Um código de correcção de erros do tipo FEC usando paridades em que cada sumário cobre exactamente um pacote anterior, que erros de omissão consegue compensar e quais os que não consegue? Justifique a sua resposta.
11. Verdade ou mentira? A técnica de correcção de erros do tipo FEC usando *interleaving* podia ser aplicada à transmissão de ficheiros. Justifique a sua resposta.
12. Qual a percentagem de informação de controlo de erros suplementar introduzida por um código de correcção de erros do tipo FEC usando *interleaving* quando cada pacote contém informação de 5 pacotes? Justifique a sua resposta.
13. Verdade ou mentira? O RTP poderia usar o campo *Timestamp* para detectar a perda de pacotes. Justifique a sua resposta.
14. Os valores do campo *Timestamp* dos pacotes RTP podem ser colocados pelo software de implementação do próprio RTP no momento em que vai enviar o datagrama UDP? Justifique a sua resposta.
15. Uma aplicação multimédia pode enviar em momentos diferentes vários pacotes RTP diferentes com o mesmo valor de *Timestamp*? Justifique a sua resposta.
16. Uma aplicação multimédia pode enviar praticamente no mesmo momento vários pacotes RTP diferentes com diferentes valores de *Timestamp*? Justifique a sua resposta.

Capítulo 10

Outros protocolos de transferência de dados

Generality in the network increases the chance that a new application can be added without having to change the core of the network.

– Autores: *M. Blumenthal e David Clark*

Os protocolos UDP e TCP foram definidos durante a década de 1980. O protocolo UDP manteve-se desde então inalterado. Em contrapartida, o protocolo TCP teve imensas evoluções, com especial realce das ligadas à problemática do controlo da saturação, um mecanismo completamente ausente na definição inicial dos protocolos TCP/IP. Durante todos estes anos a escala e a capacidade da Internet evoluíram de forma explosiva, tal como as aplicações e o tipo de tráfego dominante, com repercuções profundas nas solicitações exercidas pelas aplicações sobre os protocolos de transporte. Qual foi o impacto dessa nova realidade?

Os anos de 1990 foram dominados pelo início da comercialização da Internet e pela subida vertiginosa da sua escala e número de utilizadores. A aplicação dominante passou a ser o acesso à Web, suportada no protocolo TCP. Por isso este protocolo foi sujeito a grande pressão e o seu mecanismo de controlo da saturação continuou a ser melhorado. No advento do século XXI esta tendência manteve-se e o número de utilizadores e a capacidade dos canais continuou a subir vertiginosamente, mas juntaram-se-lhe dois factos novos.

Por um lado todas as redes convergiram para redes IP e a imagem de televisão, o vídeo e as chamadas telefónicas, primeiro de forma temerosa e mais recentemente de forma vigorosa, convergiram para a Internet. O vídeo, a televisão e as chamadas telefónicas usam, ou pelo menos usavam, principalmente UDP. A problemática da subida do tráfego UDP e da sua coexistência com o TCP subiu para primeiro plano e exerceu pressão sobre a problemática do controlo da saturação e da qualidade de serviço da rede. A qualidade de serviço da rede não é algo que se resolva exclusivamente ao nível dos protocolos de transporte, mas estes participam na solução deste problema.

A necessidade de fazer convergir as redes tradicionais com as redes IP e a Internet em geral, e a cada vez maior utilização da Internet para transportar tráfego multimédia em tempo real, levaram ao desenvolvimento de dois novos protocolos de transporte que serão a seguir introduzidos: os protocolos DCCP e SCTP.

Mais recentemente, a subida da escala da Internet foi acompanhada pela vulgarização dos dispositivos móveis. Estes dispositivos, para além de usarem canais sem fios, bastante diferentes dos canais com fios tradicionais, são igualmente caracterizados por terem várias interfaces que podem estar activas simultaneamente.

Na secção 8.5 já nos referimos ao facto do TCP ter dificuldade em lidar com a taxa de erros dos canais sem fios dado o seu impacto no controlo da saturação. Adicionalmente, tal como concebido inicialmente, o protocolo não era capaz de explorar adequadamente as novas interfaces disponíveis, em particular, o facto de que o mesmo computador poder ter várias interfaces activas simultaneamente.

O protocolo SCTP já tinha introduzido a possibilidade de cada um dos dois extremos de uma conexão terem simultaneamente diferentes endereços IP. De forma mais oportuna ainda, uma nova versão do protocolo TCP, o Multi-Path TCP, introduziu o suporte de várias interfaces e vários endereços IP de cada lado da conexão como a regra e não como exceção. Esta versão do protocolo TCP é compatível com o TCP tradicional e foi definida de tal forma que a rede, e sobretudo os equipamentos de segurança, não se apercebesse da diferença.

Com efeito, a Internet é hoje em dia sujeita a uma pressão muito significativa do ponto de vista da segurança e da escala. Essa pressão tem levado à introdução na rede de uma nova classe de equipamentos, lado a lado com os comutadores de pacotes, que analisam, filtram e bloqueiam os pacotes, na ansia de bloquearem, se possível, os potenciais atacantes. Estes equipamentos, genericamente designados por *middleboxes* (*e.g., firewalls, IDS - intrusion detection systems, etc.*), são cada vez mais numerosos e estão parametrizados por omissão para bloquearem todos os protocolos de transporte distintos do UDP e do TCP e a maioria das portas não conhecidas e normalizadas. Esta evolução traduziu-se na chamada “ossificação da Internet” que consiste, no essencial, em cada rede bloquear todos as novidades vindas das outras redes, nomeadamente por “medo do desconhecido”. A regra passou a ser “tudo o que não é explicitamente permitido, é proibido”.

Esta situação tem sido responsável pela impossibilidade de introduzir novos protocolos de transporte (e não só) na Internet, e tem sido um factor que explica o insucesso em geral de tentativas de alteração significativa dos mesmos.

Neste capítulo passamos em revista de forma muito breve algumas das propostas que têm sido feitas para alargar o número de protocolos de transporte usados na Internet global. A situação exposta acima é responsável por a maioria dos esforços de inovação a este nível estarem concentrados em propostas que mantenham a compatibilidade com o UDP e o TCP, salvo em redes particulares, controladas por uma entidade, ou por um pequeno número de entidades. Por esta razão, a grande maioria dos esforços actuais de evolução dos métodos de transferência de dados concentra-se essencialmente ao nível aplicação, como será explicado no fim do capítulo.

Começamos por apresentar o protocolo DCCP, uma evolução do UDP no sentido da introdução de controlo da saturação.

10.1 Datagram Congestion Control Protocol

Com a evolução da escala, o controlo de saturação tornou-se crítico para a Internet. O protocolo UDP não incorpora controlo da saturação, mas tornou-se a certa altura popular na Internet para o suporte do transporte de informação multimédia em tempo real, como por exemplo canais de televisão, filmes interactivos e chamadas telefónicas, ver a Secção 9.4. Quando o UDP compete pela capacidade de canais gargalo com o protocolo TCP, o resultado é muito desfavorável para o TCP. É possível introduzir controlo da saturação ao nível das aplicações, mas isso revela-se complexo e não é suficientemente geral. Procurou-se então definir um protocolo semelhante ao UDP

mas que fosse *TCP friendly*, i.e., que não monopolizasse a utilização da capacidade disponível, através da utilização de controlo da saturação.

O protocolo DCCP (*Datagram Congestion Control Protocol*) associa a capacidade de enviar datagramas, que é semelhante à do UDP, com uma noção de conexão e com controlo da saturação, incluindo suporte de ECN. O RFC 4336 e o artigo [Kohler et al., 2006] apresentam a motivação e a filosofia do protocolo, o RFC 4340 é a proposta de norma.

Apesar de à primeira vista o DCCP parecer relativamente simples, pois a sua semântica e os serviços que presta são próximos dos do UDP e, ao contrário do TCP, não assegura fiabilidade, a verdade é que o protocolo revelou-se mais complexo do que o seu objectivo inicial poderia sugerir.

Essa complexidade está relacionada com a gestão da conexão e a robustez contra ataques de negação de serviço durante a sua abertura, a robustez contra ataques ao funcionamento da conexão usando números de sequência falsos, a gestão dos números de sequência que, dada a necessidade de aceitar a perda de pacotes como fazendo parte do funcionamento normal, tornam a interpretação dos ACKs mais complexa e delicada, a gestão da mobilidade e da utilização simultânea de vários endereços distintos, e o suporte de um conjunto de diferentes algoritmos de controlo da saturação parametrizáveis pelas aplicações.

O DCCP está apenas implementado para os sistemas Linux e FreeBSD e a sua utilização fora de ambientes experimentais é praticamente nula. Provavelmente, esta situação é explicada pela resistência à introdução de novos protocolos, mas também pela generalização da difusão de canais de televisão recorrendo a UDP, serviço designado *TV-over-IP (IPTV)*, e do transporte de chamadas telefónica recorrendo a UDP, designado *Voice-over-IP (VoIP)*, mas recorrendo a uma espécie de sub-rede dedicada, controlada por um operador de acesso e com capacidade reservada para servir os seus clientes directos destes serviços. Adicionalmente, mais recentemente, a difusão de informação multimédia na Internet tornou-se mais realista usando TCP e recurso a adaptação dinâmica da resolução, ver a Secção 9.2.

10.2 Stream Control Transmission Protocol

O protocolo SCTP (Stream Control Transmission Protocol) foi definido pelo grupo de trabalho SIGTRAN (*Signalling Transport*) do IETF com o objectivo inicial de permitir a implementação da interligação de centrais telefónicas digitais sobre redes IP e para controlo e suporte de chamadas telefónicas transportadas na forma de *voice-over-IP (VoIP)*.

O RFC 3286 apresenta uma descrição introdutória e a motivação e o RFC 4960 define o protocolo. Trata-se de um protocolo muito completo que reteve do TCP o carácter orientado conexão, a fiabilidade (em opção) e o controlo da saturação, mas acrescentou-lhe novas funcionalidades entre as quais as seguintes:

4-way handshake A motivação desta nova forma de estabelecer a conexão é impedir os ataques de negação de serviço do tipo SYN-Flood, ver a Secção 7.3. As quatro mensagens necessárias para completar a abertura da conexão chamam-se INIT, INIT-ACK, COOKIE-ECHO e COOKIE-ACK. A parte que abre passivamente a conexão só a considera activa depois de receber a mensagem COOKIE-ECHO e enviar COOKIE-ACK; a parte que abre activamente a conexão só a considera activa depois de receber a mensagem INIT-ACK e enviar a resposta COOKIE-ECHO.

Mensagens (framing) O protocolo TCP é orientado a uma sequência de bytes, o SCTP, tal como o UDP, mantém a noção de mensagem o que facilita a definição

de protocolos complexos do nível superior sem recorrer a métodos especiais de delimitação das mensagens.

Multi-fluxo (*multi-streaming*) Com o protocolo TCP, para transferir diversos fluxos entre os mesmos dois interlocutores, com prioridades e papéis distintos, e sem que uns atrasem os outros, é necessário usar simultaneamente mais do que uma conexão. Se algumas delas estiverem reservadas para controlo aplicacional e com pouco tráfego, o controlo de saturação impede-as de usarem uma fracção adequada da capacidade disponível. O SCTP resolve este problema introduzindo a noção de *sub-stream* na conexão. Assim, por uma única conexão, com um mecanismo unificado de controlo da saturação, circulam diversos fluxos de mensagens, com prioridades distintas. Como resultado desta opção, a transferência de um objecto de grande dimensão também não bloqueia o envio, recepção e tratamento imediato de mensagens de controlo urgentes por exemplo.

Ordenação das mensagens (*multiple delivery mode*) Para cada fluxo é possível seleccionar um de vários modos de entrega das mensagens, nomeadamente: ordem total por fluxo, mas parcial entre fluxos, ou com entrega imediata das mensagens mal cheguem, ou seja sem ordem, como no UDP. Para emular o TCP é necessário usar um único fluxo com ordem total. Adicionalmente, o SCTP permite que um emissor avise o receptor para ignorar dados atrasados, *i.e.*, dados chegados já depois de serem úteis.

Estas diferentes formas de ordenação permitem corrigir um problema chamado “*head-of-line blocking*” que aparece quer quando um segmento TCP recebido fora de ordem não pode ser entregue devido à falta de alguns dos dados que o antecedem, quer quando dados urgentes só chegam ao receptor depois de chegarem todos os dados emitidos antes deles.

Multi-endereço (*multi-homing*) As interligações entre centrais telefónicas têm de ser mantidas com fiabilidade durante longos períodos. Por outro lado, as redes com preocupações de fiabilidade estão interligadas por múltiplos canais com o exterior. Associar diversos endereços IP ao mesmo extremo da conexão permite explorar activamente essa diversidade para efeitos de distribuição de carga e fiabilidade, sem necessidade de quebrar a conexão, e sem necessidade de estabelecer mais do que uma conexão entre as mesmas entidades. O uso desta funcionalidade para suporte directo de mobilidade tem sido também explorada.

O SCTP foi definido mais do que 20 anos depois do TCP e do UDP e por isso, para além de fornecer serviços mais ricos e diversificados do que o TCP e o UDP juntos, incorpora igualmente opções de implementação que reflectem a rica experiência que o precedeu. Um exemplo disso é a abertura da conexão que assume uma forma que permite combater ataques do tipo SYN-Flood, mas também o facto de que durante a abertura da conexão os dois interlocutores estabelecem uma *verification tag*, um número aleatório que será futuramente transmitido em todos os segmentos da conexão para combater ataques que visam introduzir segmentos falsos na mesma.

Um outro exemplo consiste na divisão de um segmento em pedaços, chamados *chunks*, que permitem o envio no mesmo segmento de dados de vários sub-fluxos mas também, os chamados *control-chunks*, que permitem bastante maior flexibilidade do que um cabeçalho com formato único, um conjunto fixo de *flags*, e um campo de opções com a dimensão máxima de 40 bytes, como os usados pelo TCP. Tal tem-se revelado insuficiente em muitas situações, sobretudo para suporte de SACKs e da opção *timestamp* simultaneamente. Existem inúmeros outros exemplos desta maturidade como por exemplo a utilização de CRCs de 32 bits, que podem ser calculados em software, ou o facto da utilização de SACKs ser obrigatória e não uma opção.

O protocolo SCTP está disponível em todos os sistemas de operação e existe bastante experiência operacional da sua utilização [Dreibholz et al., 2011]. Apesar disso, a sua utilização pelas aplicações mais populares não é generalizada, em particular pelos

clientes e servidores Web que poderiam beneficiar muito das suas funcionalidades para o acesso a serviços Web. Isso está, provavelmente, ligado ao problema de todos os protocolos distintos de TCP e UDP serem na prática bloqueados, mas também ligado ao facto de que a adopção de SCTP implica a utilização de uma nova interface de sockets SCTP, o que implica que muitas aplicações teriam de ser alteradas.

Recentemente, o SCTP foi introduzido na chamada “*WebRTC – Real-Time Communications for the Web*” [Jennings et al., 2013], uma iniciativa para dotar os diferentes *browsers Web* de um conjunto de interfaces, mecanismos e protocolos que facilitam a utilização de comunicações interactivas, com suporte de tráfego multimédia directamente entre *browsers Web*. No entanto, para assegurar que o tráfego SCTP não é bloqueado, a proposta WebRTC prevê que os segmentos SCTP sejam encapsulados em datagramas UDP, ver o RFC 6951.

A seguir vamos ver outro protocolo que propõe uma forma alternativa de lidar com alguns dos problemas que o SCTP já tenta resolver, nomeadamente a utilização de diversos canais.

10.3 Multi-Path TCP

Como foi referido no início do capítulo, a multiplicidade de novos dispositivos móveis como *smart-phones, pads, etc.* introduziu uma nova categoria de computadores que podem ligar-se à rede através de diversos canais (*e.g.*, interfaces de rede Wi-Fi, interfaces de redes celulares, interfaces de rede *ethernet* com fios, ...). Cada uma dessas interfaces tem um endereço IP distinto. Adicionalmente, esses dispositivos são móveis e a mobilidade implica frequentemente a alteração do endereço IP.

As conexões TCP tradicionais estão associadas aos endereços IP e às portas dos extremos. Qualquer alteração de um endereço implica a quebra da conexão. O protocolo Multi-Path TCP (MPTCP) procura responder a este problema permitindo que uma conexão TCP possa explorar diversos canais e diversos endereços IP entre os dois extremos da conexão. Desta forma é possível explorar vários canais e caminhos alternativos entre os dois extremos da conexão para distribuição de carga. Adicionalmente, é possível ter diversas associações activas entre diferentes pares de endereços IP, e um dispositivo móvel manter uma conexão TCP activa mesmo que o dispositivo altere o endereço IP e o ponto de ligação à rede de uma das suas interfaces. Tudo o que é necessário é que durante a transição pelo menos alguma associação entre um par de endereços continue disponível para comunicar.

O MPTCP foi desenhado tendo em consideração algumas das razões que levaram à adopção residual do DCCP e em parte do SCTP: serem protocolos diferentes de TCP e do UDP e implicarem alterações à interface de sockets e às aplicações [Paasch and Bonaventure, 2014]. Assim, a sua concepção teve por objectivos: ser capaz de usar simultaneamente diversas interfaces e caminhos dentro da rede, coexistir facilmente com TCP (ser *TCP friendly*) e usar os mesmos tipos de controlo de saturação que o TCP, não necessitar de alterações das aplicações e, ao competir com TCP, continuar a existir equidade entre conexões.

O MPTCP aparece para o TCP convencional como uma nova opção. Se ambas as extremidades da conexão suportarem MPTCP, passam a usar uma noção de sub-fluxo, cada um dos quais associado a um par distinto de endereços, mas com as mesmas portas [Raiciu et al., 2012]. O estabelecimento da conexão inicial estabelece o primeiro sub-fluxo mas, posteriormente, outros podem ser acrescentados ou suprimidos. No estabelecimento da conexão inicial as duas extremidades também trocam chaves que lhes permitem depois identificar inequivocamente a conexão independentemente dos sub-fluxos. Cada um dos sub-fluxos é acrescentado por um novo *three-way handshake* suplementar, cujo estabelecimento, devido a usar endereços distintos, é sujeito a autenticação usando as chaves trocadas no estabelecimento da conexão inicial.

Para manter a equidade com as conexões TCP, o MPTCP executa controlo de saturação independente em cada um dos seus sub-fluxos, mas globalmente coordenado, e altera o débito usado em cada um deles para explorar da melhor forma possível a capacidade disponível. Para manter a equidade, o protocolo usa um controlo da saturação que se traduz numa subida mais lenta da dimensão da janela de emissão em cada um dos sub-fluxos do que o TCP tradicional no seu único fluxo, mas que é globalmente equitativa para o MPTCP. Adicionalmente, o MPTCP privilegia o uso dos sub-fluxos menos saturados [Wischik et al., 2011]. Esta é uma área onde o MPTCP abriu, provavelmente, novas avenidas para a investigação no controlo da saturação multi-caminho.

Seguindo uma tendência moderna, e a exemplo do protocolo DCCP, os algoritmos de controlo da saturação são vários e podem ser alterados pelas aplicações.

A possibilidade de usar simultaneamente diversos sub-fluxos através de interfaces distintas permite usar a conectividade de uma interface para compensar os erros noutra, ou activar dinamicamente novas interfaces durante os períodos de transição entre pontos de acesso à rede. Por exemplo, a ligação via uma rede celular pode compensar momentaneamente as falhas de conectividade durante a transição da ligação via Wi-Fi entre diferentes pontos de acesso [Paasch et al., 2012]. A utilização simultânea de várias interfaces foi também usada num centro de dados para demonstrar que o MPTCP pode ser usado para que uma conexão MPTCP única possa explorar simultaneamente todos os caminhos disponíveis na rede do centro de dados entre dois servidores interligados simultaneamente por múltiplos canais [Raiciu et al., 2011].

O MPTCP introduz mecanismos sofisticados e inovadores de melhoramento do desempenho e da longevidade das conexões TCP, através da utilização simultânea de vários sub-fluxos que terminam em interfaces distintas, e introduz novas formas de gerir o controlo da saturação. Para além destes aspectos, a definição e implementação do protocolo inclui um notável trabalho de engenharia para assegurar que os seus segmentos não são bloqueados pelas *middleboxes* hoje mais comuns na Internet. O MPTCP está disponível para Linux, Android, FreeBSD e iOS desde a versão 7. O RFC 6824 introduz a definição do protocolo.

Para terminarmos esta breve discussão de outros protocolos de transporte alternativos ao UDP e ao TCP, vamos a seguir voltar a analisar os factores que têm contribuído para o seu relativo insucesso, assim como algumas aproximações alternativas ao problema.

10.4 *Middleboxes* e “ossificação” da Internet

O nível rede em geral e a Internet em particular, tal como os temos apresentado até aqui, são infra-estruturas que transportam pacotes de dados entre computadores, sem os modificar ou bloquearem. A verdade é hoje em dia mais complexa pois no interior da rede, para além dos comutadores de pacotes, existem numerosos equipamentos que intervêm no encaminhamento dos pacotes, podendo bloqueá-los ou modificá-los.

Esses equipamentos podem ser de vários tipos, nomeadamente: *firewalls*, *stateful firewalls*, *deep-packet inspection firewalls*, *network address translation boxes*, *proxies*, ... etc. Os *firewalls* bloqueiam pacotes que não pertencem a origens, destinos, protocolos e portas permitidos dentro da rede que protegem. São equipamentos que no essencial bloqueiam fluxos de pacotes com base nos endereços IP e portas origem / destino.

Os outros tipos de *firewalls* fazem uma análise mais fina dos pacotes e bloqueiam pacotes que não pertencem a conexões previamente estabelecidas, ou que contêm números de sequência impossíveis, ou bloqueiam opções não desejadas dos protocolos, ou ainda que procuram no conteúdo dos pacotes dados que revelam a presença de potenciais vírus ou outras formas de ataque implementadas a nível aplicacional.

As *network address translation (NAT) boxes* são equipamentos, muito comuns nas redes domésticas e nas redes dos operadores celulares, que transformam os endereços origem dos pacotes, permitindo que um ou poucos endereços IP sejam partilhados por um número muito maior de computadores para acesso à Internet. A forma como estes equipamentos funcionam exige o reconhecimento dos protocolos de transporte usados no *payload* dos pacotes IP, o que restringe a utilização de protocolos de transporte aos nossos bem conhecidos TCP e UDP.

Os *proxies* são equipamentos que terminam as conexões noutros computadores diferentes dos originalmente requeridos, e que partem as conexões TCP em diferentes conexões para melhorarem o desempenho.

Este equipamento são hoje em dia muitíssimo numerosos. De acordo com algumas estatísticas recenseadas em [Raiciu et al., 2013] a partir de diferentes estudos, cerca de 90% dos utilizadores domésticos e cerca de 80% dos operadores celulares utilizam equipamentos do tipo NAT. A maioria dos operadores celulares utilizam *firewalls*, assim como muitos computadores individuais e servidores. Todas as redes empresariais utilizam *middleboxes* em número quase tão significativo como o dos comutadores de pacotes, e um número relevante de redes institucionais e de redes de acesso usam *proxies*.

Como resultado desta situação, o protocolo IP já não é o gargalo da pilha de protocolos da Internet, ver a Secção 4.4. Este gargalo compreende actualmente também os protocolos UDP e TCP como está ilustrado na Figura 10.1.

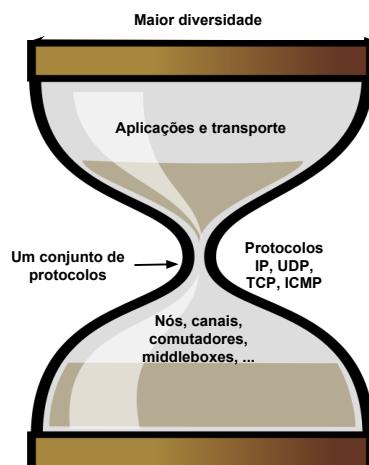


Figura 10.1: Os protocolos IP, UDP e TCP são o novo gargalo da ampulheta; eles separam as aplicações da “diversidade mitigada” da rede

Este fenómeno é em geral designado por “ossificação” da Internet e, como vimos através das funcionalidades que se tentaram introduzir no transporte de dados (múltiplos fluxos, múltiplas semânticas de fiabilidade e ordenação, múltiplas formas de realizar o controlo de saturação, exploração de múltiplas interfaces, *etc.*) e do insucesso da sua adopção, constatamos que estamos numa situação contraditória e que tende para uma grande rigidez de alternativas de transporte e de formas diferentes de suportar as necessidades do desenvolvimento das aplicações.

A visão inicial existente na comunidade que desenvolvia aplicações para a Internet, e que foi transmitida na primeira parte deste livro, é que as aplicações eram desenvol-

vidas usando um conjunto de protocolos de transporte, implementados directamente pelos sistemas de operação. Na verdade, a noção de transporte evoluiu e tornou-se mais rica. Hoje em dia a maioria das aplicações não usa directamente a interface de sockets, mas antes um conjunto de bibliotecas e APIs que fornecem um conjunto de serviços especialmente concebidos para um dado conjunto de aplicações.

Esta nova visão das funcionalidades de transporte permite que as mesmas mas carem a forma como os serviços de transporte são de facto disponibilizados. Muitos serviços de “transporte de dados” são hoje em dia implementados sobre protocolos, inclusive de nível aplicacional [Raiciu et al., 2013]. Para alguns, o protocolo HTTP é praticamente o único meio de transporte de informação universal actualmente disponível.

10.5 Propostas para aprofundamento

Ao longo deste capítulo foram indicadas várias referências bibliográficas que permitem o aprofundamento do estudo das diversas facetas nele tratadas. Segue-se um conjunto de propostas de trabalhos de aprofundamento dessas facetas.

1. Realize, sob a forma de um relatório, um estudo mais aprofundado do protocolo DCCP utilizando as referências que foram indicadas acima.
2. Realize, sob a forma de um relatório, um estudo mais aprofundado do protocolo SCTP. Utilize a referência [Budzisz et al., 2012] como um ponto de partida suplementar para além das que foram indicadas acima.
3. Realize, sob a forma de um relatório, um estudo mais aprofundado do protocolo MPTCP utilizando as referências que foram indicadas acima.
4. Realize, sob a forma de um relatório, um estudo mais aprofundado sobre a evolução recente dos protocolos de transporte fiável de informação partindo da referência [Raiciu et al., 2013].