jlegatheaux Adicionar arquivos via upload ...

2 contribuidores  

Cru Culpa

331 linhas (221 sloc) 28,5 KB

# RC 2020/2021 - Tarefa 1

## Metas

Com esta tarefa, os alunos obterão uma melhor compreensão de como as redes de comutação de pacotes funcionam, qual é o tempo de trânsito dos pacotes neste tipo de rede e a forma de computá-lo, e como as propriedades da rede de comutação de pacotes impactam o desempenho de ponta a ponta de envio de informações de um nó para outro.

## Suposições

A seguir consideraremos, por hipótese, que todos os links são perfeitos e nunca pacotes corrompidos ou perdidos. A mesma propriedade se aplica aos nós - eles são à prova de balas e nunca travam ou perdem pacotes. Além disso, como estamos usando o CNSS, todos os cálculos são executados instantaneamente, sem qualquer atraso.

As configurações de rede dos diferentes experimentos feitos nesta atribuição usam enlaces com as mesmas características: largura de banda de 2 M bits por segundo, ou 2.000.000 bps, e um tempo de propagação, ou latência, de 20 ms, uma vez que possuem 4.000 Km cada (4000 Km / 200.000 Km por segundo = 4 x 10 ^ 3/2 x 10 ^ 5 Km por segundo = 2 x 10 ^ -2 s = 20 ms).

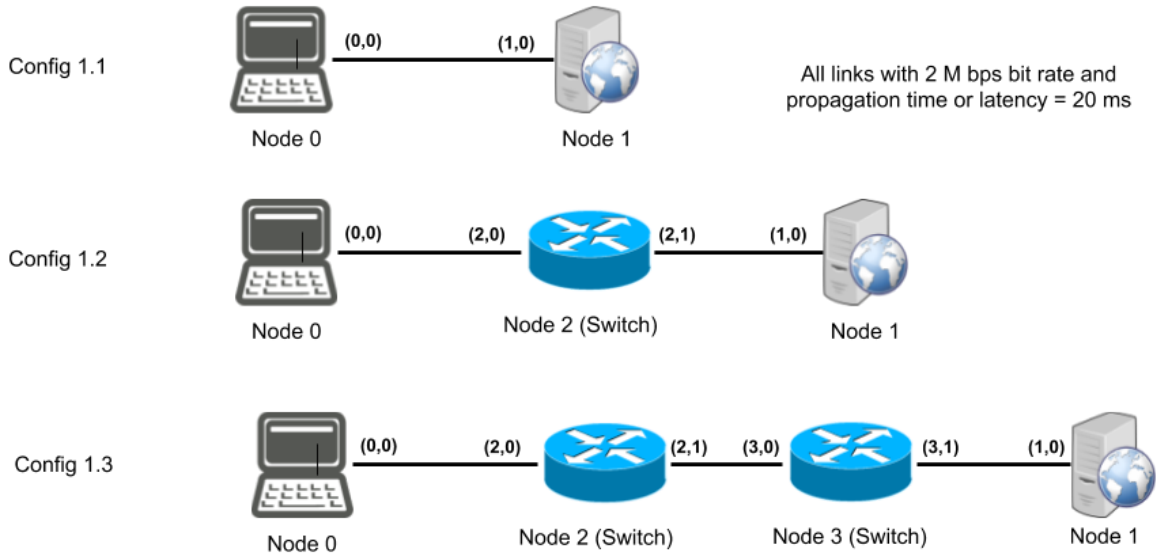
## Compreender armazenamento e encaminhamento, tempo de trânsito de ponta a ponta e o tempo necessário para transferir informações em uma rede comutada por pacote

**AVISO 1: por favor, estude as seções 3.2 e 3.3 do livro de apoio ao curso para entender completamente esta seção.**

**AVISO 2: não se esqueça de atualizar o CNSS para sua versão mais recente.**

A seguir, faremos diversos experimentos com o objetivo de entender quais fatores contribuem para o tempo de trânsito de um pacote em uma rede, bem como ter uma primeira noção do que impacta o tempo necessário para transferir informações de um nó. para um diferente em uma rede de comutação de pacotes.

As três configurações de rede usadas no primeiro conjunto de experimentos configs / config1.1.txt , configs / config1.2.txt e configs / config1.3.txt são representadas na figura abaixo:




.

O objetivo é calcular o tempo necessário para transferir um arquivo do **nó 0** para o **nó 1** , usando duas soluções diferentes nessas três configurações de rede diferentes.

Com a primeira solução de transferência de arquivo, o **nó 0** envia o arquivo enviando **10** (é fácil mudar esse número) pacotes de 10.000 bytes, consecutivamente, para o **nó 1** . Com a segunda solução, uma carga útil de 10 x 10.000 bytes é enviada em um único (grande) pacote.

O código do aplicativo do **nó 0** , a parte do remetente, está no arquivo filesSender.java . Este nó da aplicação envia em sequência e **imediatamente** durante a sua inicialização, 10 pacotes de 10.000 bytes para o **nó 1** . O número de pacotes a enviar pode ser facilmente alterado, mas, salvo indicação em contrário, sempre enviaremos 10 pacotes quando o arquivo for enviado usando vários pacotes. Mais tarde, após 60 segundos, o nó 0 envia o mesmo arquivo em um único pacote.

```
public class filesSender extends AbstractApplicationAlgorithm {

        público estático int BLOCKSIZE = 10000 ; // 10000 * 8 = 8
        público estático int TOTAL_PACKETSIZE = BLOCKSIZE + Pacote . HEADERS


        int totSent = 0 ;
        int totalBlocks = 0 ;
        tamanho do arquivo int = 0 ;

        public filesSender () {
                super ( true , " remetente de arquivos " );
        }

        public int initialise ( int now , int node_id , Node self , String [
                super . inicializar (agora, node_id, self, args);
                log ( 0 , " começando " );
                if (args . length ! = 1 ) {
```

```
                        Sistema . err . println ( " remetente de arquivos: hora
                        Sistema . saída ( - 1 );
                }
                totalBlocks =  Inteiro . parseInt (args [ 0 ]);
                para ( int i =  1 ; i <= totalBlocks; i ++ ) {
                        eu . enviar (self . createDataPacket ( 1 , novo  byte [ E
                        log (agora, " pacote enviado de tamanho " + TOTAL_PACKETS
                }
                eu . set_timeout ( 60000 ); // 60 segundos depois
                return  0 ;
        }

        public  void  on_timeout ( int  agora ) {
                eu . send (self . createDataPacket ( 1 , novo  byte [ TOTAL_PACKE
                log (agora, " pacote enviado de tamanho: " + TOTAL_PACKETSIZE * t
        }

        public  void  on_receive ( int  now , DataPacket  p ) {
                log (agora, " ack recebido " + p +  " c / carga útil " + nova  St
        }

        public  void  showState ( int  now ) {
                 System . para fora . println (nome +  " enviado "  + totSent +
        }

}
```

O código do remetente merece apenas alguns comentários. Um ***tempo limite*** é configurado durante a inicialização para definir um alarme para 60.000 ms ou 60 segundos depois. Assim, o envio do arquivo em um único pacote, um grande pacote, é executado pelo upcall `on_timeout()` . Para tornar as duas soluções de transferência comparáveis, quando um único pacote é enviado, seu tamanho foi adicionado de n vezes o tamanho do cabeçalho de um pacote (n vezes 20 bytes). Portanto, o número total de bytes transferidos com as duas soluções difere apenas em 20 bytes (o tamanho do cabeçalho do pacote grande). O código do aplicativo do nó emissor está preparado para receber um ack do receptor. No entanto, com esses três experimentos, nenhum acks é enviado pelo nó 1, o receptor. O aplicativo usa o registro para mostrar como a transferência está progredindo. O código do aplicativo do nó receptor também usa o registro para mostrar quando os pacotes são recebidos.

A única parte do aplicativo receptor que merece ser mostrada, veja o arquivo `filesReceiver.java` , é seu `on_receive()` upcall:

```
    public  void on_receive ( int now, DataPacket p) {
            totReceived ++ ;
            log (agora, " obteve: "  + p +  " n. " + totRecebido);
    }
```

que apenas registra a recepção dos pacotes e o número total de pacotes recebidos até o momento de cada recepção.

Para completar os experimentos, você deve baixar o diretório `assignment1` e tê-lo abaixo do diretório de nível superior do código-fonte, geralmente `src` , uma vez que todas as fontes Java `assignment1` pertencem ao pacote `package assignment1` .

Para executar o experimento 1.1, supondo que o diretório atual está acima do `src/` diretório (que contém `cnss/` e `assignment1/` diretórios) e que existe um `bin/` diretório lado a lado do `src` diretório (contendo os executáveis), o seguinte comando faz o trabalho:

```
java -cp bin cnss.simulator.Simulator src/assignment1/configs/config1.1.txt
```

Ao seguir o resultado da simulação é fácil reconhecer que o 10º pacote foi recebido no tempo 420, portanto, o arquivo levou 420 ms segundos para ser transferido com a solução que enviou pacotes de 10 vezes 10.000 bytes, e também levou 420 ms para ser transferido em um único grande pacote de 100.000 bytes mais 200 bytes de cabeçalhos.

Você pode calcular facilmente esses resultados analiticamente. Para entender completamente como isso deve ser feito, você deve estudar as seções 3.2 e 3.3 do Capítulo 3 do livro do curso. O tempo de transmissão (Tt) de um pacote com 10.000 bytes (80.000 bits) para um link com taxa de bits de 2 Mbps é de 40 ms (Tt = tamanho em bits / taxa de bits = 80000/2000000 = 0,040). O tempo de transmissão do pacote grande é 10 vezes maior e o tempo de propagação do link é de 20 ms.

Depois de entender tudo, você pode agora prosseguir para a próxima experiência e dar o seguinte comando:

```
java -cp bin cnss.simulator.Simulator src/assignment1/configs/config1.2.txt
```

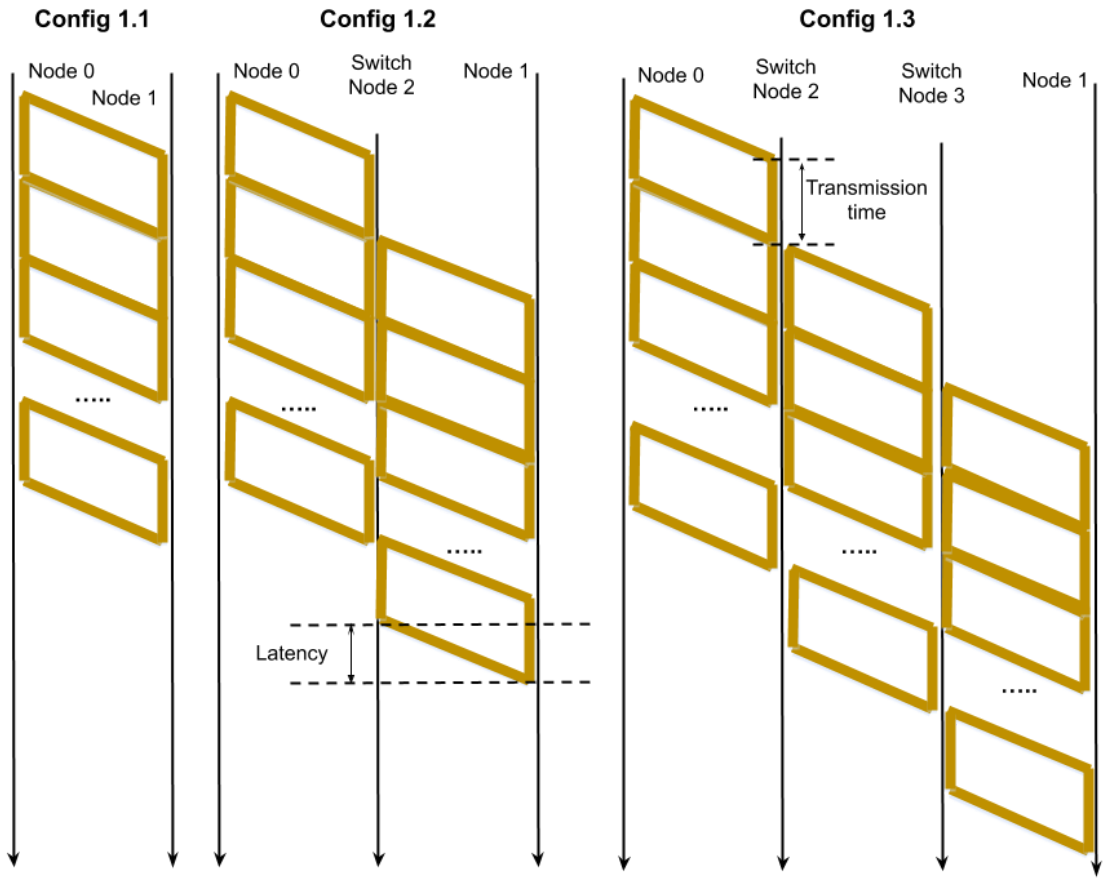
Você também pode executar o terceiro experimento emitindo o mesmo comando com um arquivo de configuração diferente:

```
src/assignment1/configs/config1.3.txt
```

Agora as coisas ficam mais interessantes, já que o tempo que as duas transferências levaram são diferentes. É importante notar que na Internet, assim como em quase todas as redes, não é possível enviar pacotes tão grandes quanto o grande pacote usado para enviar o arquivo em um único pacote. NCSS não faz restrições ao tamanho dos pacotes porque é uma ferramenta de simulação. No entanto, os experimentos também mostram que não é muito interessante usar pacotes de tamanhos grandes (este é um conceito relativo relacionado à largura de banda dos links), como veremos a seguir.

De fato, no experimento 2, quando o arquivo é enviado em 10 pacotes, leva 480 ms para chegar ao destino, em vez de 420 no primeiro experimento, enquanto leva 540 ms no terceiro experimento. Você deve repetir a análise analítica necessária para entender por que esses resultados são obtidos. Do experimento 1 para 2, o tempo de transferência aumentou 60 ms, enquanto no experimento 3 aumentou 60 ms sobre o experimento 2 e 120 ms sobre o experimento 1.

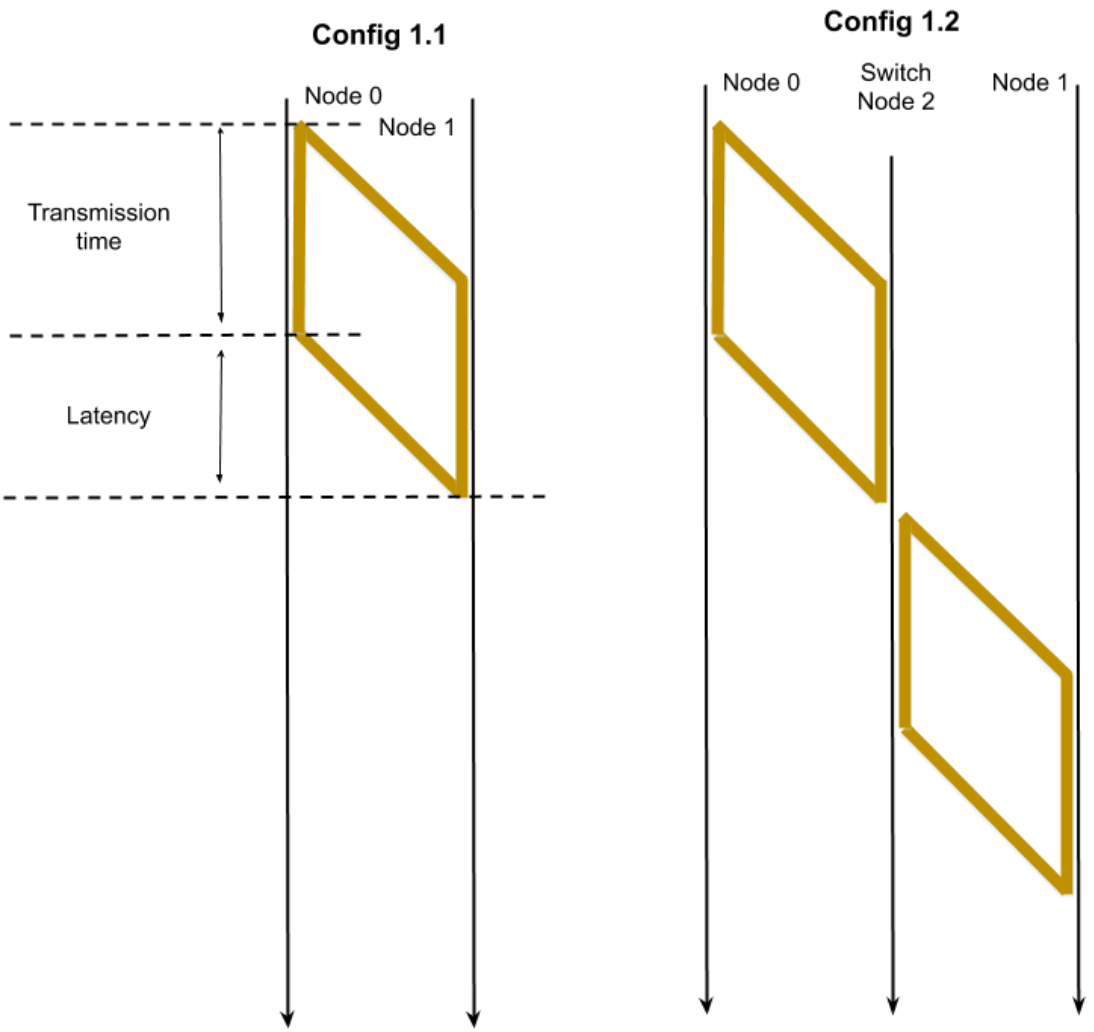
O aumento de um experimento para o seguinte está relacionado ao tempo de transmissão extra introduzido pelo switch e link extras, adicionado à latência do link extra. Você pode usar a figura abaixo para entender melhor os motivos que o explicam.




.

Como é explicado no livro, as redes de comutação de pacotes empregam switches que usam o princípio **store & forward** , que afirma que os pacotes devem ser totalmente recebidos por um switch antes de serem encaminhados para o próximo (ou para um sistema final). Embora o switch possa enviar e receber vários pacotes em paralelo em links diferentes, cada pacote só pode ser encaminhado após ser totalmente recebido, analisado e processado. Só então é que a interface de saída em seu caminho para o destino é escolhida e sua transmissão pode prosseguir. Assim, se substituirmos um link por vários links interconectados por switches, mesmo que a soma das latências dos novos links seja igual à latência do substituído, cada link introduz um tempo de transmissão extra para o pacote ponta a ponta tempo de transferência.

Se olharmos agora para os resultados dos três experimentos no que diz respeito à transferência do arquivo em um único grande pacote, mais lições podem ser aprendidas. No experimento 1, a transferência usando um único pacote levou 420 ms para ser concluída, o mesmo quando vários pacotes foram usados. No entanto, no experimento 2, a mesma transferência leva 840 ms e 1260 ms no experimento 3. De um experimento para o seguinte, o tempo de transferência aumentou 420 ms. Esse aumento também se deve ao mesmo motivo, um tempo de transmissão mais a latência do switch extra e do link introduzido a cada vez. No entanto, agora, o tempo de transmissão do pacote grande leva 400 ms em vez dos 40 ms que cada pacote "pequeno" levava. Figura




ilustra claramente a diferença entre os experimentos 1 e 2. A lição é, se os links têm taxas de bits que introduzem tempos de transmissão significativos, aumentar o tamanho dos pacotes pode introduzir aumentos inesperados no tempo de trânsito.

Antes de prosseguir, você deve revisar os 3 experimentos e pegar uma folha de papel e refazer os cálculos nos três casos para comparar seus cálculos com os resultados mostrados pelas simulações. Você deve estar convencido no final de que calcular os tempos de transferência em uma rede onde nenhum pacote é perdido e não há tráfego competitivo (outras fontes enviando pacotes que cruzam os mesmos links que seus pacotes também cruzam) é muito fácil.

Você também pode repetir os mesmos experimentos com arquivos maiores (mais pacotes) ou com links com taxas de bits mais altas. Se você aumentar a largura de banda dos links de 2 Mbps para, por exemplo, 100 Mbps ou 1000 Mbps (1 Gbps), os tempos de transmissão se tornarão muito pequenos. Por exemplo, o envio de 80.000 bits a 1 Mbps requer 80 ms, enquanto o envio do mesmo pacote a 1 Gbps requer apenas 0,08 ms ou 80 micro segundos. À medida que a largura de banda aumenta, o fator dominante no tempo de trânsito de ponta a ponta é a latência dos links.

**Aviso:** ao alterar um arquivo de configuração, você deve prestar atenção ao fato de que você não pode cometer erros ou de outra forma o NCSS travar miseravelmente. Em particular, cada token no arquivo deve ser separado do nex por exatamente um caractere de espaço e você não deve inserir linhas apenas com espaços. Em uma próxima versão, melhoraremos as habilidades de análise de arquivos de configuração do CNSS.

Uma última observação interessante a respeito do CNSS está relacionada ao fato de que esses três arquivos de configuração não possuem `stop parameter`. Às vezes, não é necessário introduzir um, pois o CNSS reconhece que nenhum outro evento pode ser disparado na simulação e interrompe seu processamento.

## Transmissão de dados com controle de fluxo

Em redes reais, vários problemas podem surgir, como pacotes sendo perdidos ou entregues fora de serviço aos nós receptores. As soluções dos experimentos anteriores não podem lidar com esses problemas do mundo real e irão falhar, uma vez que os dados enviados e recebidos podem ser diferentes. Na tarefa 2, estudaremos métodos para lidar com essas características das redes reais.

Adicionalmente, as soluções de transferência de dados anteriormente apresentadas também não podem resolver outro problema, relacionado, não com a rede, mas com as características dos nós do mundo real, nomeadamente, o facto de as suas capacidades de processamento serem diferentes. Portanto, um nó muito poderoso pode enviar dados a uma taxa que um receptor menos poderoso não é capaz de processar em tempo hábil. Se for esse o caso, os pacotes também podem ser perdidos porque o receptor não pode processar todos os pacotes que recebe, e a única solução é descartar alguns deles. O resultado final é o mesmo como se esses pacotes descartados não fossem entregues pela rede.

As soluções para este problema são conhecidas como soluções de ***adaptação de fluxo ou de controle de fluxo*** , que fornecem métodos para adaptar a taxa de envio dos emissores à taxa de processamento dos receptores.

Existe outro problema de adaptação de taxa relacionado ao fato de que um remetente de alta capacidade pode saturar uma rede incapaz de entregar pacotes enviados a uma taxa muito alta. Por exemplo, ao mesmo tempo, outros nós também estão enviando muitos pacotes que cruzam os mesmos links que nosso remetente de alto desempenho. Nesse caso, também é necessário adaptar a taxa dos nós emissores à capacidade disponível na rede. A solução para esse problema é chamada de ***controle de congestionamento da rede*** .

Os métodos de controle de fluxo e os métodos de controle de congestionamento de rede são diferentes, mas ambos compartilham algumas características comuns. Na verdade, ambos podem depender de sinais enviados por receptores (ou pela rede) aos remetentes, dizendo-lhes para parar, recusando o envio de pacotes ou para continuar enviando-os. Ambos os métodos são discutidos em vários capítulos do livro, a saber, os capítulos 6, 7 e 8. A esta altura, você não precisa estudar esses capítulos para entender esta tarefa, mas precisará deles para as próximas tarefas.

## O controle de fluxo de parada e espera

O método mais simples de controle de fluxo é conhecido como controle de fluxo "Stop & Wait" ou S&W. S&W também é o nome do protocolo que depende desse método. É um protocolo muito simples. Cada vez que o remetente enviar um pacote, ele entrará em uma fase de espera, até a recepção de um sinal do receptor que significa que recebeu o pacote e está pronto para o próximo. Esses pequenos pacotes de sinal são conhecidos como pacotes de confirmação ou pacotes ACK.

O algoritmo de aplicação de um nó receptor usando este protocolo está disponível em arquivo `FilesReceiverAck.java` . Novamente, o único upcall que vale a pena discutir é `on_receive()` esse.

```
    public  void on_receive ( int now, DataPacket p) {
        totReceived ++ ;
        log (agora, " obteve: "  + p +  " n: " + ( int ) (p . getPayload () [ 0 ]))
        eu . send (self . createDataPacket (p . getSource (), ( " ack " + totReceiv
    }
```

Ele registra a recepção do pacote e responde ao remetente enviando um pacote ACK com a quantidade de pacotes recebidos até o momento (incluindo este).

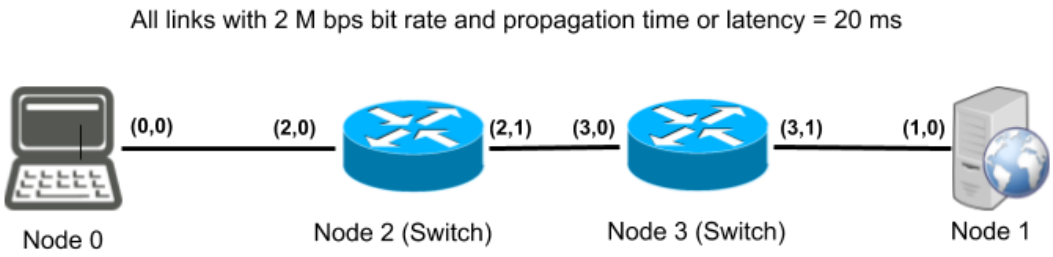
O código do nó emissor é um pouco mais elaborado. Ele está contido em um arquivo `NaifSwSender.java` e é mostrado a seguir (mostramos apenas os métodos `initialise()` , `on_receive()` e `showState()` ). Chamamos essa solução e as próximas de ***Naif,*** pois não são aceitáveis para cenários do mundo real onde as redes podem perder pacotes.

```
public  class  NaifSwSender  extends  AbstractApplicationAlgorithm {

.....

        public  int  initialise ( int  now , int  node_id , Node  self , String [
                 super . inicializar (agora, node_id, self, args);
                if (args . length ! =  1 ) {
                         Sistema . err . println ( " remetente de arquivos: hora
                        Sistema . saída ( - 1 );
```

```
                }
                totalBlocks =  Inteiro . parseInt (args [ 0 ]);
                log ( 0 , " começando " );
                startTime = now;
                totSent =  1 ;
                byte [] pl =  novo  byte [ BLOCKSIZE ];
                pl [ 0 ] = ( byte ) (totSent &  0xff );
                eu . enviar (self . createDataPacket ( 1 , pl));
                log (agora, " pacote enviado de tamanho " + TOTAL_PACKETSIZE + "
                return  0 ;
        }


        public  void  on_receive ( int  now , DataPacket  p ) {
                log (agora, " ack packet: " + p + " pl: " + new  String (p . getF
                if (totSent <= totalBlocks -  1 ) {
                        totSent ++ ;
                        byte [] pl =  novo  byte [ BLOCKSIZE ];
                        pl [ 0 ] = ( byte ) (totSent &  0xff );
                        eu . enviar (self . createDataPacket ( 1 , pl));
                        log (agora, " pacote enviado de tamanho " + TOTAL_PACKETS
                } else  if (totSent == totalBlocks) {
                        transferTime = now - startTime;
                        totBytesTransferred =  TOTAL_PACKETSIZE * totalBlocks;
                        float transferTimeInSeconds = ( float ) transferTime /  1
                        e2eTransferRate = ( int ) (totBytesTransferred * 8  / tra
                        log (agora, totBytesTransferred + " bytes transferidos em
                }
        }

        public  void  showState ( int  now ) {
                 System . para fora . println (nome +  " enviado "  + totSent +
                Sistema . para fora . println (name + "  " + totBytesTransferred
                                + transferTime + " ms em " + e2eTransferRate + "
        }

}
```

O remetente, após a inicialização de suas variáveis, envia o primeiro pacote. Então, cada vez que recebe um pacote do receptor (um ACK), ele prossegue para o próximo pacote (embora nem todos os pacotes tenham sido enviados e confirmados). Quando o último ACK é recebido, ele calcula a taxa de transferência e o imprime. Para aumentar a clareza dos logs (nada mais), ele coloca no primeiro byte do pacote enviado sua ordem (assim, se mais de 254 pacotes forem enviados, esse número mudará para 0 novamente).

Todos os experimentos a seguir são realizados usando a configuração de rede abaixo, que já é conhecida.




Agora é interessante experimentar a transferência do arquivo com esta nova solução. Isso é feito facilmente usando a configuração `config.1.4.txt` e realizando o experimento 4 com o seguinte comando:

```
java -cp bin ... .Simulator src/assignment1/configs/config1.4.txt
```

O resultado do experimento mostra que a transferência do arquivo levou 2.400 ms, muito mais do que nos experimentos anteriores, a uma taxa ponta a ponta de 334.000 bits por segundo ou cerca de 1/3 da largura de banda dos links.

É interessante explicar esses resultados ao mesmo tempo que os calcula analiticamente. Você também deve calcular o que é chamado de ***taxa*** de ***uso*** dos links. Se você tiver dúvidas, o capítulo 6 do livro pode ajudá-lo.

Outro experimento com o mesmo protocolo e configuração de rede é o experimento 5, que usa o arquivo e o comando de configuração:

```
java -cp bin ... .Simulator src/assignment1/configs/config1.5.txt
```

durante o qual um arquivo com tamanho de 100 pacotes, em vez de 10, é transferido. O resultado, como esperado, mostra uma taxa de transferência ponta a ponta idêntica. Este experimento será usado como referência para os próximos experimentos.

## Outra Solução para Simulação do Protocolo S&W com CNSS

O CNSS possui um recurso que permite que o algoritmo de aplicação de um nó execute uma etapa de processamento periódica. Isso pode ser conseguido tornando o método `initialise()` para retornar o período de execução periódica necessário. Por exemplo, se esse método retornar 1, o nó pode executar uma ação toda vez que o relógio avançar. Cada vez que o relógio marca, o kernel do nó chama o upcall `on_clock_tick()` onde se pode indicar o que deve ser executado periodicamente.

O arquivo `NaifSwSenderP.java` contém outra simulação CNSS do protocolo S&W. Com essa solução, mostrada abaixo, apenas alguns métodos são importantes. O `initialise()` método, não mostrado, é muito simples, pois apenas inicializa o processo e todas as variáveis solicitadas. No final retorna 1 para exigir a execução do upcall `on_clock_tick()` cada vez que o relógio avança. Também não mostramos o `showState()` método, pois é idêntico à solução anterior. Os únicos upcalls realmente importantes são aqueles mostrados.

```
public  void on_clock_tick ( int now) {
                if (mayProceed && totSent < totalBlocks) {
                        totSent ++ ;
                        byte [] pl =  novo  byte [ BLOCKSIZE ];
                        pl [ 0 ] = ( byte ) (totSent &  0xff );
                        eu . enviar (self . createDataPacket ( 1 , pl));
                        log (agora, " pacote enviado de tamanho " + TOTAL_PACKETS
                        mayProceed =  false ;
                }
        }

        public  void on_receive ( int now, DataPacket p) {
                log (agora, " ack packet: " + p + " pl: " + new  String (p . getP
                mayProceed =  true ;
                if (totSent == totalBlocks) {
                        transferTime = now - startTime;
                        totBytesTransferred =  TOTAL_PACKETSIZE * totalBlocks;
                        float transferTimeInSeconds = ( float ) transferTime /  1
                        e2eTransferRate = ( int ) (totBytesTransferred * 8  / tra
                        log (agora, totBytesTransferred + " bytes transferidos em
                }
        }
```

O `on_clock_tick(int now)` upcall executa um ***padrão*** regular : ele testa as condições antes de realizar alguma ação (as pré-condições da ação). Se essas condições forem atendidas, ele executa alguma ação, caso contrário, não faz nada. Pode haver mais de uma ação, cada uma com sua própria pré-condição. No exemplo apresentado, a pré-condição é

```
mayProceed && totSent < totalBlocks
```

ou seja, existem blocos de dados ou pacotes para enviar (ainda não enviamos todos) e nada nos impede de enviar. Se essa pré-condição for válida, um pacote é enviado e a variável `mayProceed` passa a ser falsa, já que o remetente entra no estado de espera, aguardando o recebimento do ACK (o "pare de transmitir e espere" em nome de S&W do protocolo). Na verdade, essa variável registra o estado de execução do protocolo. Se for o caso `true`, o remetente pode enviar um pacote, caso contrário, deve aguardar que esse estado se torne `true`, ou seja, "mais um pacote pode ser enviado se houver um disponível".

O `on_receive(...)` upcall processa os ACKs recebidos. Ele registra a recepção e muda o estado para `mayProceed = true` uma vez que o remetente pode prosseguir. Se for o último ACK, ele calcula a taxa de transferência.

Esse padrão de algoritmo de aplicativo é adequado para expressar algoritmos como autômatos. Os upcalls `on_clock_tick()`, `on_receive()`, `on_timeout()` etc., podem todos ser estruturada como um tipo de "interruptor" sobre as condições de estado (ou estado autómato), executar as acções correspondentes pacotes de estado e possivelmente enviar, e, finalmente, actualizar o estado. Esse tipo de padrão é útil para estruturar certos algoritmos de protocolo melhor do que com uma seqüência serial de séries de ações. Com esse padrão, esses algoritmos no CNSS são executados como uma sucessão de etapas de processamento, cada uma composta por ações relacionadas ao estado atual, seguida por uma mudança de estado, e então aguardam a próxima etapa de processamento ou evento. É o que se denomina estrutura de programa semelhante a processamento de eventos.

No CNSS, esses autômatos nunca terminam e podem executar muitas ações vazias se usarem muito pequenos `clock_ticks`. Pequenos períodos de relógio podem ser necessários para responder a mudanças de estado em tempo hábil. No mundo real, essa seria uma solução que desperdiça ciclos de CPU, entretanto, como se trata apenas de uma simulação, o padrão pode ser implementado conforme ilustrado, sem inconvenientes muito graves. O parâmetro `stop` pode ser usado para evitar que a simulação nunca termine. O único efeito colateral bizarro é o aviso mostrado pelo CNSS:

```
warning - 1 events not run; stoped too early?
```

porque o próximo tique do relógio não foi executado. Também é possível parar a simulação chamando `System.exit()` um upcall, mas isso deve ser considerado uma má prática, pois é uma parada abrupta e pode impedir o processamento de alguns eventos programados para acontecer posteriormente.

O experimento número 6 corresponde à transferência de um arquivo de 100 pacotes do nó 0 para o nó 1 usando a `NaifSwSenderP.java` implementação do protocolo S&W. Você pode executá-lo emitindo o comando

```
java -cp bin ... .Simulator src/assignment1/configs/config1.6.txt
```

É interessante notar que esta solução leva 1 milissegundo a mais que a anterior. Você pode explicar isso? Você pode pensar em quando o primeiro pacote foi enviado e se leva mais tempo para receber um ACK e enviar o próximo pacote.

Para praticar esse novo padrão, você deve usá-lo nas próximas entregas.

# Entrega pela primeira vez - protocolos fixos de envio de janela

Você notou que o protocolo S&W apresentou, em experimentos anteriores, uma taxa de transferência ponta a ponta inferior à capacidade dos links. Isso pode ser melhorado usando os chamados ***protocolos de janela deslizante*** ; consulte o capítulo 6 do livro se quiser estudá-los com antecedência. Com o protocolo S&W, um único pacote pode estar em trânsito do emissor para o receptor, ou seja, enviado e ainda não confirmado. Os protocolos de janela deslizante permitem que um determinado número (N) de pacotes, com N ≥ 1, seja enviado pelo remetente com antecedência. N é chamado de tamanho da janela e pode ser definido como o número de pacotes em trânsito do emissor para o receptor, ou seja, pacotes já enviados, mas ainda não confirmados.

Por exemplo, com uma janela de tamanho 2, ou seja com N = 2, dois pacotes podem ser enviados antecipadamente, e sempre que um ACK for recebido, mais um pacote pode ser enviado, etc. Também é possível implementar a mesma ideia enviando 2 pacotes, um após o outro em sequência, esperando por seus 2 ACKs e, em seguida, transmitindo mais 2 pacotes e, em seguida, reiniciando depois de receber mais 2 ACKs, etc. No entanto, a maneira anterior como descrevemos o algoritmo pode ter mais desempenho do que o último em certas condições (esta é uma questão difícil de ser entendida) e essa é a preferida.

O algoritmo do remetente `NaifWindSender.java` deve ser chamado e deve ter dois parâmetros: o tamanho do arquivo, fornecido como o número de pacotes a enviar, e o tamanho da janela. Deve ser estruturado da mesma forma que o algoritmo `NaifSwSenderP.java` e usar os mesmos métodos para calcular a taxa de transferência de ponta a ponta e o mesmo método `showStatus()` (contido no arquivo `NaifSwSender.java` ). O receptor é o que está no arquivo `FilesReceiverAck.java` .

Os experimentos 7, 8 e 9 (veja o conteúdo de seus arquivos de configuração) contêm os 3 experimentos que você deve executar com sucesso. Arquivos `results/config1.7.txt`, `results/config1.8.txt` e `results/config1.9.txt` conter o registro resulta a sua solução deve produzir. Sua solução estará correta se produzir os mesmos resultados que os contidos nesses arquivos. Você pode testá-lo usando, por exemplo, um programa `diff` para comparar sua saída (armazenada em um arquivo) com esses arquivos.

Os instrutores também podem testar sua solução com parâmetros diferentes dos apresentados acima. Deve estar preparado para discutir os resultados de cada experiência, nomeadamente os valores das taxas de transferência end-to-end obtidas em cada caso e compará-los com os resultados do protocolo S&W.

# Segunda entrega - protocolos de tamanho de janela de aumento dinâmico

O algoritmo de janela deslizante anterior usa uma janela de envio de tamanho constante. Em certas circunstâncias, o remetente não tem informações sobre o tamanho correto a ser utilizado. Uma solução pode ser começar com uma taxa de transmissão muito baixa e então, se tudo estiver bem, aumentá-la. Alguns desses algoritmos de janela dimensionados dinamicamente podem ser caracterizados como algoritmos de aumento aditivo ou algoritmos de aumento exponencial. O algoritmo de aumento aditivo adiciona 1 ao tamanho da janela quando todos os pacotes da janela anterior foram confirmados. Algoritmos que aumentam exponencialmente dobram o tamanho da janela quando todos os pacotes da janela anterior foram confirmados.

Por exemplo, na primeira rodada, um pacote é enviado e seu ACK recebido. Na rodada seguinte, 2 pacotes são enviados e 2 ACKs são recebidos. No próximo, 4 pacotes são enviados e 4 ACKs são recebidos, e assim por diante. Na rodada K, os `2^(K-1)` pacotes são enviados e o mesmo número de ACKs recebidos. Você deve implementar este algoritmo exato.

O algoritmo do remetente deve ser chamado `NaifMincSender` e tem um parâmetro, o número de pacotes a enviar, ou seja, o tamanho do arquivo. Deve ser estruturado da mesma forma que o algoritmo `NaifSwSenderP.java` e usar os mesmos métodos para calcular a taxa de transferência de ponta a ponta, assim como o método `showStatus()`. O receptor é o mesmo, o que está no arquivo `FilesReceiverAck.java`.

O experimento 10 (veja o conteúdo de seu arquivo de configuração) contém a configuração do experimento que você deve ser capaz de executar. O arquivo `results/config1.10.txt` contém os resultados do log que sua solução deve produzir. Sua solução está correta se produzir os mesmos resultados. Você pode testá-lo usando, por exemplo, um programa `diff` para comparar sua saída (armazenada em um arquivo) com o arquivo `results/config1.10.txt` .

Os instrutores também podem testar sua solução com parâmetros diferentes dos apresentados acima. Você deve estar preparado para discutir os resultados do seu experimento, ou seja, os valores da taxa de transferência de ponta a ponta, e compará-los com os resultados do protocolo S&W.