Giovane Ribeiro

Laboratório de Redes de Computadores 3

INF01154 - Redes de Computadores N

Turma D

Universidade Federal do Rio Grande do Sul - UFRGS

Instituto de Informática

Engenharia de Computação

Brasil

2015, 25 de Setembro

1 Gerador de Tráfego UDP

1.1 Ambientes

O programa foi desenvolvido e testado em dois sistemas operacionais diferentes. Uma máquina rodando Fedora Linux 22 e em uma maquina virtual rodando Windows 7. Através dos testes são evidenciadas pequenas diferenças entre a execução dos programas nos diferentes ambientes.

1.2 Código Fonte

O código fonte disponibilizado na disciplina foi alterado de modo a ser um gerador de tráfego utilizando pacotes UDP. Primeiramente foi inserida a opção de se entrar os dados da carga de tráfego desejada coma opção "-r n" 1.1, sendo n o número de Kbps desejados.

O algoritmo utilizado calcula o número de pacotes necessários para atingir a carga desejada. Esse cálculo é feito com base no tamanho do buffer somado aos bytes extras que são necessários para se formar um pacotes UDP/IP e por fim transformando a carga desejada de segundos para milisegundos. A taxa entrada é transformada de Kbps para bps, logo o tamanho do pacote é calculado somando-se o tamanho do buffer com o tamanho dos cabeçalhos UDP e IP. O tamanho do buffer utilizado foi de 750 Bytes, esse valor foi escolhido através de testes onde obteve uma maior estabilidade comparado a outros tamanhos de buffer. Por fim a taxa encontrada esta em segundos, logo uma transformação para milisegundos é necessária, pois este foi o intervalo utilizado pela função sleep() que foi escolhido arbitrariamente por ser pequeno o suficiente para não gerar rajadas e manter uma estabilidade razoável no tráfego gerado. O código do cálculo para o envio da carga pode ser visto em 1.2

A função sleep() tem uma descrição diferente em cada um dos sistemas 1.3. No Windows a função sleep recebe um valor em milisegundos, já no Linux é necessário utilizar a função usleep() para obter-se 100 milisegundos de intervalo. É importante ressaltar que o sistema principal utilizado para desenvolvimento e testes foi o Fedora Linux, e

```
1  case 'r':
     i++;
3    r = atoi(argv[i]);
     break;
```

Listing 1.1 – Opção para Inserir taxa de Envio

```
    // Transforma o taxa de entrada de Kbits para bits
    tx = r * 1024;
    // Calcula o tamanho do pacote pelo tamanho do buffer + UDP/IP overhead
    ps = (8 * buffer_size) + 224;
    // encontra o numero de pacotes por segundo que devem
    //ser enviados para satisfazer a carga desejada
    n = tx / ps ;
    // Transforma a carga desejada de segundos para miliseguntos
    n = (n / 10) ;
```

Listing 1.2 – Cálculo Para Envio de Pacotes

```
#ifdef _WIN32
    // Sleep por 1 milisegundo
    Sleep(100);
#else
    // Sleep por 1 milisegundo
    usleep(100000);
#endif
```

Listing 1.3 – Sleep de 1 milisegundo

posteriormente adaptado para Windows.

Também é necessário lembrar que para compilar o código no CodeBlocks no Windows é necessário inserir a opção "-std=gnu99" em "Sttings->Compiler->Other Options". Já no sistema Linux utilizando o GCC a compilação decorre normalmente com o make disponibilizado nos exemplos de aula.

1.3 Testes

Foram efetuados testes com alguns valores diferentes escolhidos arbitrariamente. Também é possível uma comparação entre os dois sistemas através dos resultados obtidos. Os valores utilizados foram 2000, 1000, 500 e 676 kbps, sendo 676 a multiplicação de 26*26(minha idade) por ser um trabalho individual.

Os primeiros testes com r=2000 podem ser vistos na Figura 1 e na Figura 2. Como é possível visualizar no Linux a taxa média é de 2,01 Mbps e no Windows de 1,82 Mbps. Também é possível notar que no Windows existe uma instabilidade maior no tráfego gerado.

Para r=1000 podem ser vistos na Figura 3 e na Figura 4. Como é possível visualizar no Linux a taxa média é de 1,01 Mbps e no Windows de 912,4 Kbps. Para r=500 Figura 5 e Figura 6 obtém-se uma taxa média de 505 Kbps no Linux e 456,2 no Windows.

Por fim com r=676 obtém-se uma pequena variação do valor desejado mas ainda assim satisfatório como pode ser visto na Figura 7 e na Figura 8 onde a taxa média no Linux é de 696 Kbps e no Windows de 627 Kbps. Também é possível notar que no

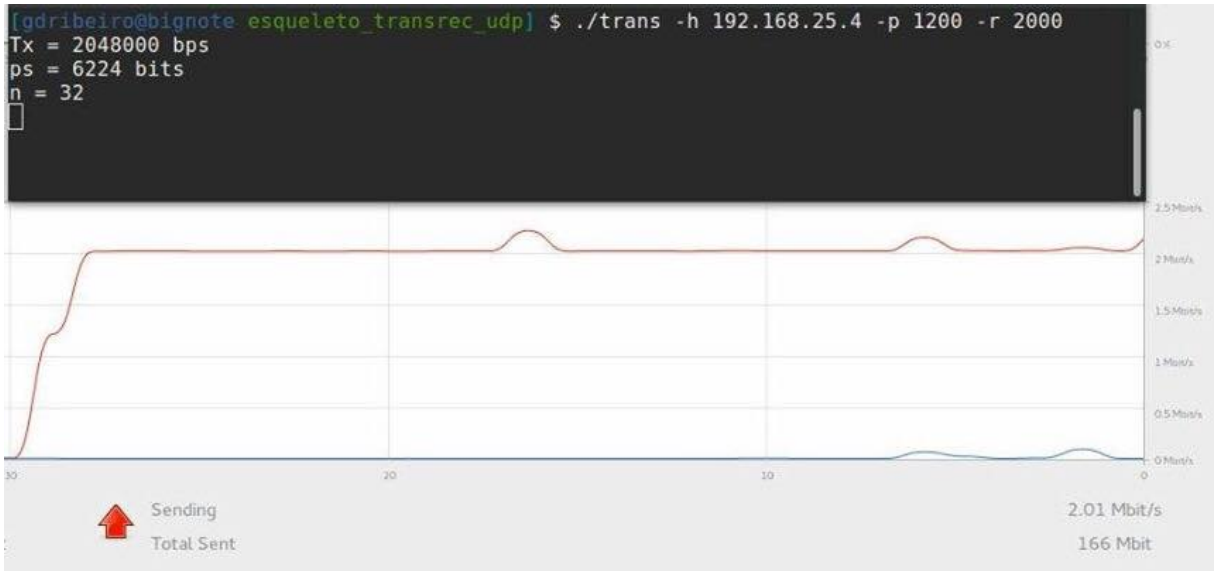

Figura 1 – Tráfego no Linux para r = 2000 Kbps

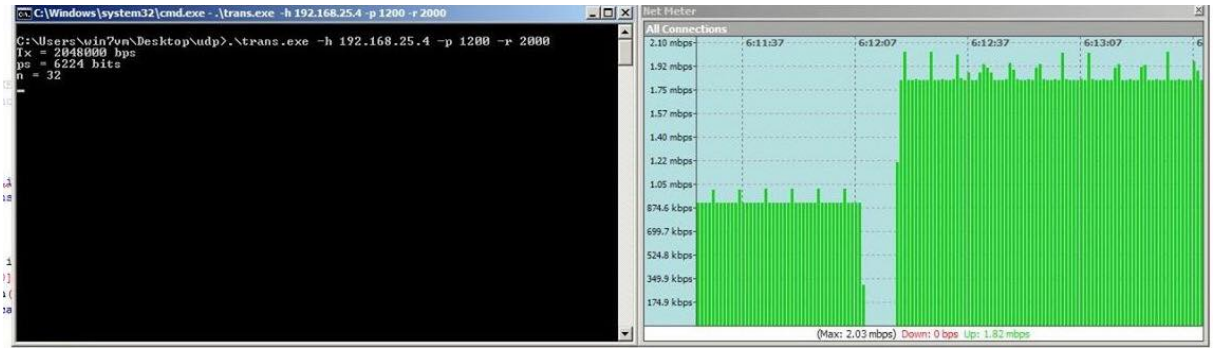

Figura 2 – Tráfego no Windows para r = 2000 Kbps

Windows existe uma instabilidade maior no tráfego gerado além de uma imprecisão maior do que apresentado no sistema Linux.

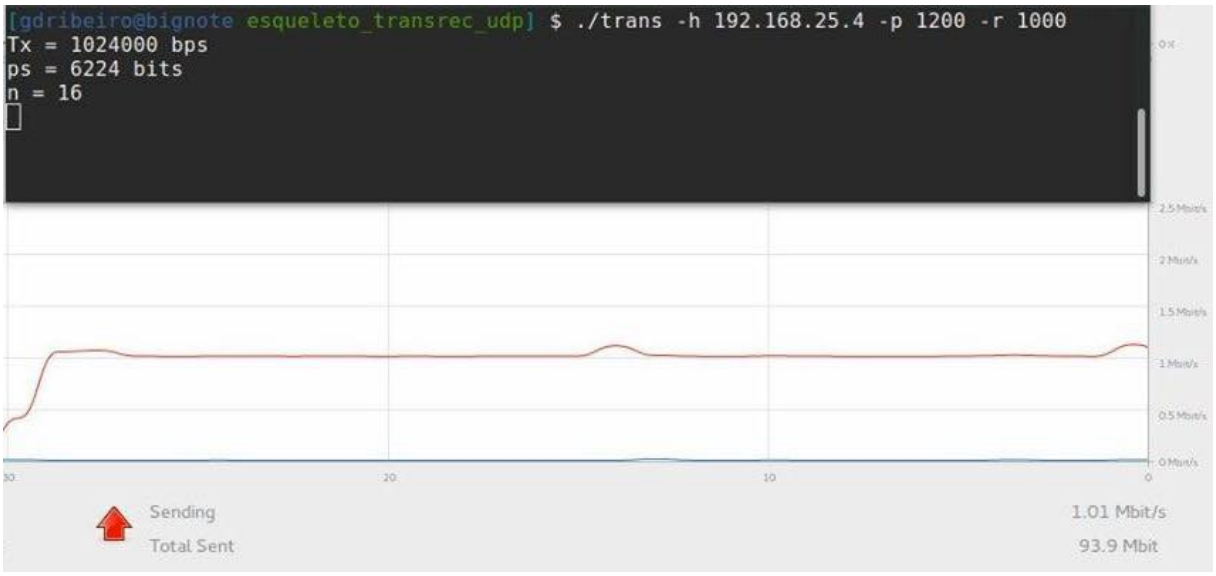

Figura 3 – Tráfego no Linux para r = 1000 Kbps

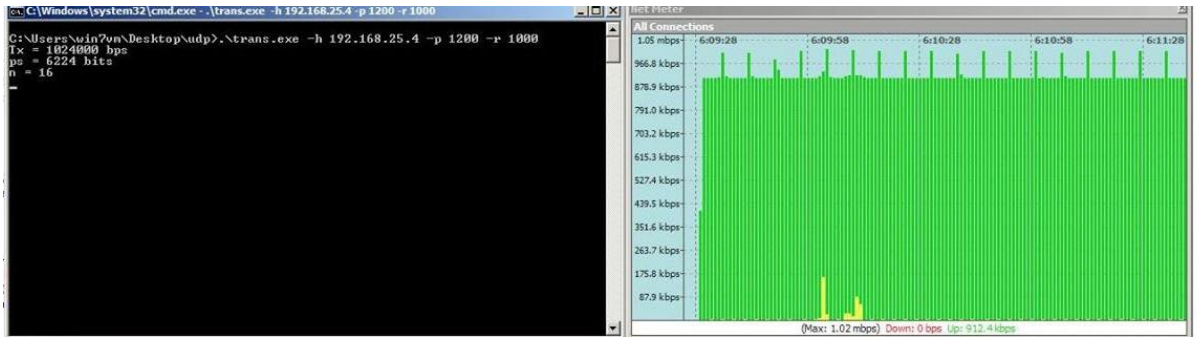

Figura 4 – Tráfego no Windows para r = 1000 Kbps

Figura 5 – Tráfego no Linux para r = 500 Kbps

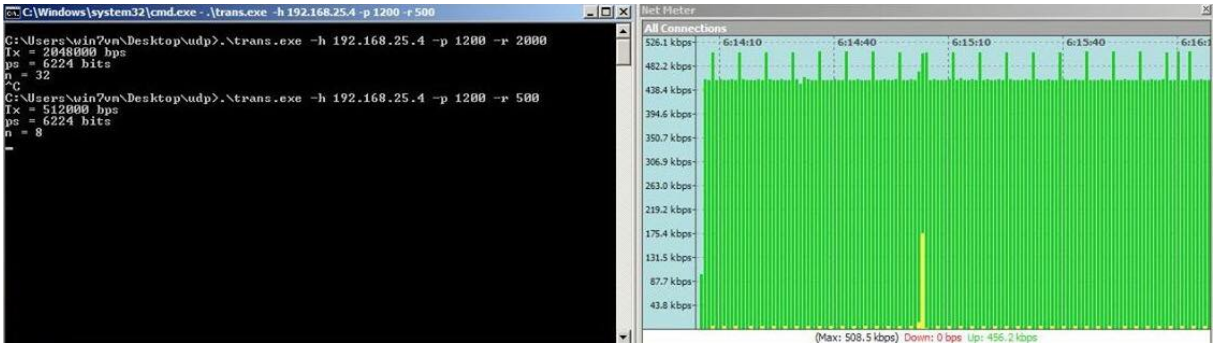

Figura 6 – Tráfego no Windows para r = 500 Kbps

Figura 7 – Tráfego no Linux para r = 676 Kbps

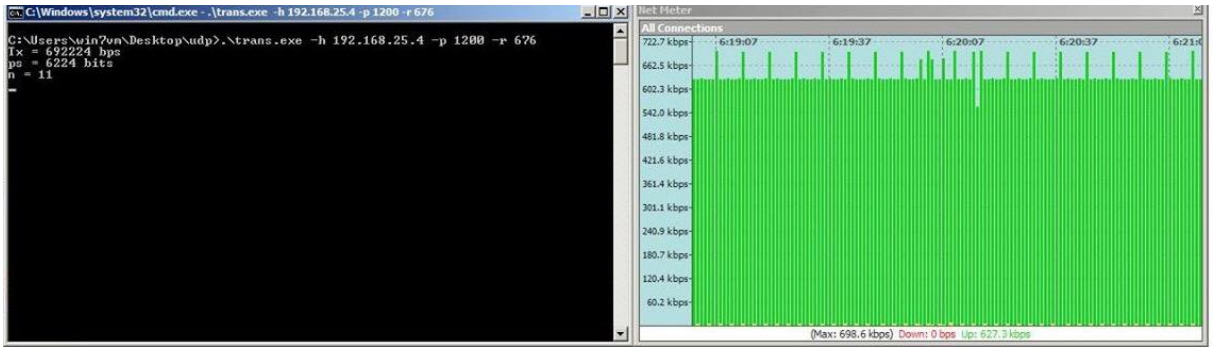

Figura 8 – Tráfego no Windows para r = 676 Kbps

2 TCP

2.1 Ambiente

Os programas presentes no diretório esqueleto_tcp disponibilizado nas disciplina foram analisados e alterados de acordo com os objetivos. Todo o desenvolvimento assim como os testes foram realizados em uma máquina rodando Fedora Linux 22 foram compilados utilizando GNU GCC.

2.2 Código Fonte

O código foi alterado em ambos os programas visando a eficiência e também para facilitar os testes realizados.

2.2.1 Servidor

O código fonte do servidor foi alterado principalmente para inserir a porta como parâmetro no momento de execução ao contrário de definido estaticamente no código fonte. Isso permite uma que sejam executados servidores na mesma máquina em portas diferentes de uma forma mais rápida e evitando um número maior de compilações e versões diferentes do mesmo programa. Dessa forma a linha de comando para executar o servidor é "$ ./srv -p 2023", inserindo-se o parâmetro -p e o número da porta logo em seguida. O código inserido para esse fim é mostrado em 2.1.

2.2.2 Cliente

As principais alterações no código do cliente foram para monitorar o tempo e salvar o log. As principais variáveis inseridas são vistas em 2.2 com os comentários necessários. Também foi necessário inserir uma variável pthread_t para executar a thread que monitora

```
1  case 'p': // porta
     i++;
3    PORTA_SRV = atoi(argv[i]);
     if(PORTA_SRV < 1024) {
5    printf("Valor da porta invalido\n");
     exit(1);
7    }
     break;
```

Listing 2.1 – Opção de Porta para o Servidor

```
// Usadas para calcular o tempo
struct timespec startCount, stopCount;
// usada para calcular o numero de pacotes enviados
unsigned int i;
// Mutex para garantir exclusao mutua na hora do calculo
// do tempo para nao alterar i e nem stopCount no momento do calculo.
pthread_mutex_t blockSending;
// Arquivo para salvar as medias por segundo.
FILE *logFile;
// Funcao executada na thread usada para fazer os calculos e
// monitorar o tempo em paralelo
// Retirando tempo de dentro do laco de execucao do envio de pacotes
// Tornanco assim um pouco mais precisas as medidas
void *getRate(void *arg);
```

Listing 2.2 – Variávies Globais Necessárias

o tempo e salva o log da taxa de transmissão e uma variável para entrar o nome do arquivo no qual o log deve ser salvo.

Foi retirado do código o bind() referente ao cliente, pois definir uma porta para o cliente não é uma função crítica nesse caso, logo essa atividade é deixada a cargo do sistema operacional para atribuir uma porta qualquer ao socket.

Na função main() foi alterada principalmente da seguinte forma 2.3 para realizar as funções desejadas. Primeiramente inicia-se a thread responsável pelo monitoramento do tempo de envio e cálculo da taxa de envio, a cada segundo o log e salvo no arquivo. Também inicia-se em "i=0"para contar o número de pacotes e efetuar o cálculo da taxa de envio. O Clock é setado para o inicio. No laço temos a região crítica do envio do pacote e do incremento da variável "i", assim como a função que salva o valor final do relógio, usado para o cálculo do tempo. Logo após essas execuções o mutex é liberado. Também ao final do código o arquivo onde o log é salvo é fechado.

A função void *getRate(void *arg) 2.4 é executada na thread para efetuar o monitoramento do tempo de 1 segundo entre os logs e também salvar o log no arquivo. A cada vez que o relógio atinge 1 segundo a a taxa de envio é calculada e salva no arquivo de log, também a variável i e os relógios são reiniciados. É importante notar que o código dentro do if é executado de maneira atômica e é protegido pelo mutex.

2.3 Testes

Todos os teste foram realizados em uma LAN residencial conectada por um roteador. No momento dos testes somente as duas máquinas utilizadas para os testes estavam conectadas ao roteador e não havia nenhuma outra maquina ou equipamento conectado tanto através de cabo ethernet ou através de wifi.

```
// Inicia thread para monitorar o tempo e salvar o log
  int thd = pthread_create(&countTimeThread, NULL, getRate, NULL);
  // Seta i=0 para contar o numero de pacotes enviados
  i = 0;
  // Seta o tempo de inicio para o calculo da taxa de envio
  clock_gettime(CLOCK_REALTIME, &startCount);

  // Loop that sends to the server
  while(1)
  {
    pthread_mutex_lock(&blockSending);
    // Send a buffer of 1250 bytes to the server over the TCP connection
    if (send(s, (const char *)&str, sizeof(str),0) < 0 ) {
      printf("Error during transmission!\n");
      close(s);
      break;
    }
    //send(s, (const char *)&str, sizeof(str),0);
    clock_gettime(CLOCK_REALTIME, &stopCount);
    i++;
    pthread_mutex_unlock(&blockSending);
  }
  fclose(logFile);
  return 0;
}
```

Listing 2.3 – Funçao main()

```
void *getRate(void *arg)
{
  while(1)
  {
    // Usa 1 segundos para o calculo da taxa de envio
    if ((stopCount.tv_sec - startCount.tv_sec) > 0){
      pthread_mutex_lock(&blockSending);
      fprintf(logFile, "%lu\n", (unsigned long int) (i * 1250 * 8));
      i = 0;
      clock_gettime(CLOCK_REALTIME, &stopCount);
      clock_gettime(CLOCK_REALTIME, &startCount);
      pthread_mutex_unlock(&blockSending);
    }
    fflush(logFile);
  }
}
```

Listing 2.4 – Função getRate Executada pela Thread

Foram realizados duas metodologias de testes diferentes. Na primeira um processo foi executado por 1 minuto e logo após outro processo foi iniciado e também executaram juntos por 1 minuto, depois com 3 processos e então com 4. Desse teste o primeiro log contém todas as alterações devido ao inicio e fim de execução dos outros processos. Estes arquivos de log estão na pasta "teste_1"que acompanha o material do trabalho. A outra metodologia utilizada foi gerar um log por cada execução com um número diferente de processos. Um log para 1 processo, outro para dois processos, outro para 3 e outro para 4. Nesse teste os processos que executaram juntos foram também iniciados juntos. Os logs dos testes executados com a segunda metodologia estão na pasta "teste_2".

Durante os testes a utilização da rede permaneceu como mostrado na Figura 9, onde mostra a máquina que executava os clientes.

Figura 9 – Utilização da Rede

2.3.1 Teste 1

No teste 1 o arquivo de log "log_total.txt"contém todos as alterações de tráfego que houveram devido ao início da execução dos outros processos. A quantidade de tráfego enviada quando somente um processo estava em execução foi em torno de 93,1 Mbps. Com dois processos em execução a quantidade de tráfego ficou em torno de 45 Mbps e 46 Mbps. Com três processos em execução a velocidade variou entre 30 Mbps e 34 Mbps, mantendo-se mais próxima de 30 Mbps. Já no último caso com quatro processos o tráfego permanesceu em torno de 22 Mbps e 23 Mbps.

A Figura 10 mostra o grafico do tráfego pelo tempo. Desde o início até em torno de 60 segundos somente um processo está ativo então outro processo é iniciado, no gráfico é possível visualizar a queda da taxa de transmissão do processo devido ao controle de tráfego implementado pelo protocolo TCP. Isso também acontece em torno de 120 segundos quando o terceiro processo entra em execução. E também em 180 segundos quando temos 4 processos em execução. As medidas apresentam oscilações mas é possível visualizar que a capacidade do canal é dividida entre o número de processos ativos. O crescimento no final do gráfico é devido ao termino da execução dos outros processos.

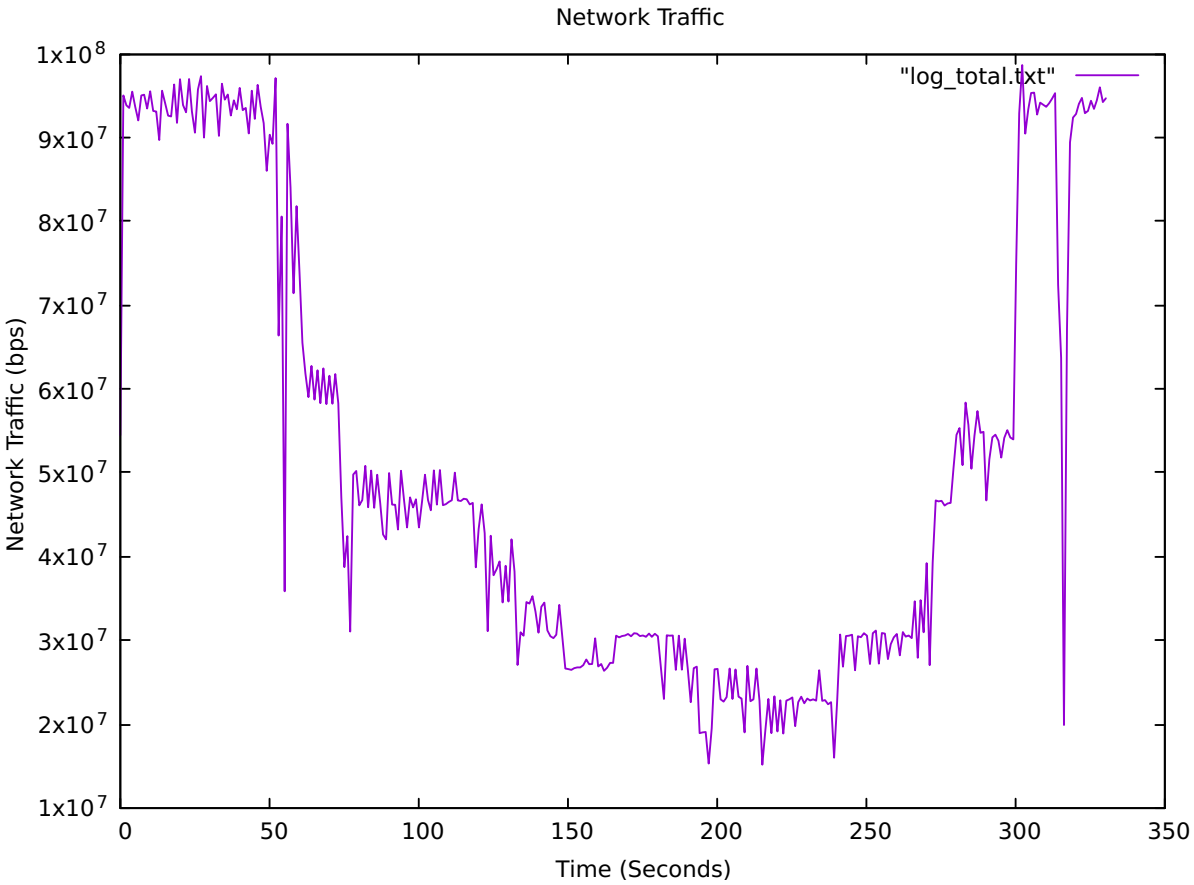

Figura 10 – Gráfico De Tráfego Total com Diferentes Números de Processos

2.3.2 Teste 2

No segundo teste cada teste com um número diferente de processos foi executado de maneira separada, colocando em execução os processos ao mesmo tempo, diferentemente da primeira metodologia de teste. Os logs do teste 2 são encontrados na pasta "teste_2".

A Figura 11 mostra o gráfico para somente um processo em execução. Nota-se que o processo utiliza toda a capacidade do canal. Porém nos outros gráficos apresentados na Figura 12, Figura 13 e Figura 14, a divisão da capacidade da rede não é bem distribuida entre os processos. Essa situação é vista principalmente com dois e com três processos. No entanto com quatro processos a divisão é mais justa entre os processo utilizando a rede.

2.3.3 Conclusão

No teste 1 foi possível verificar uma estabilidade e justiça maior na divisão da capacidade de transmissão do canal do que evidenciado no teste 2. Mas Em ambos é evidenciado que o protocolo TCP/IP implementa um controle na utilização da rede, onde existe uma justiça para a divisão da capacidade de transmissão de maneira dividir igualmente a capacidade do canal entre os processos e/ou máquinas que estão utilizando a

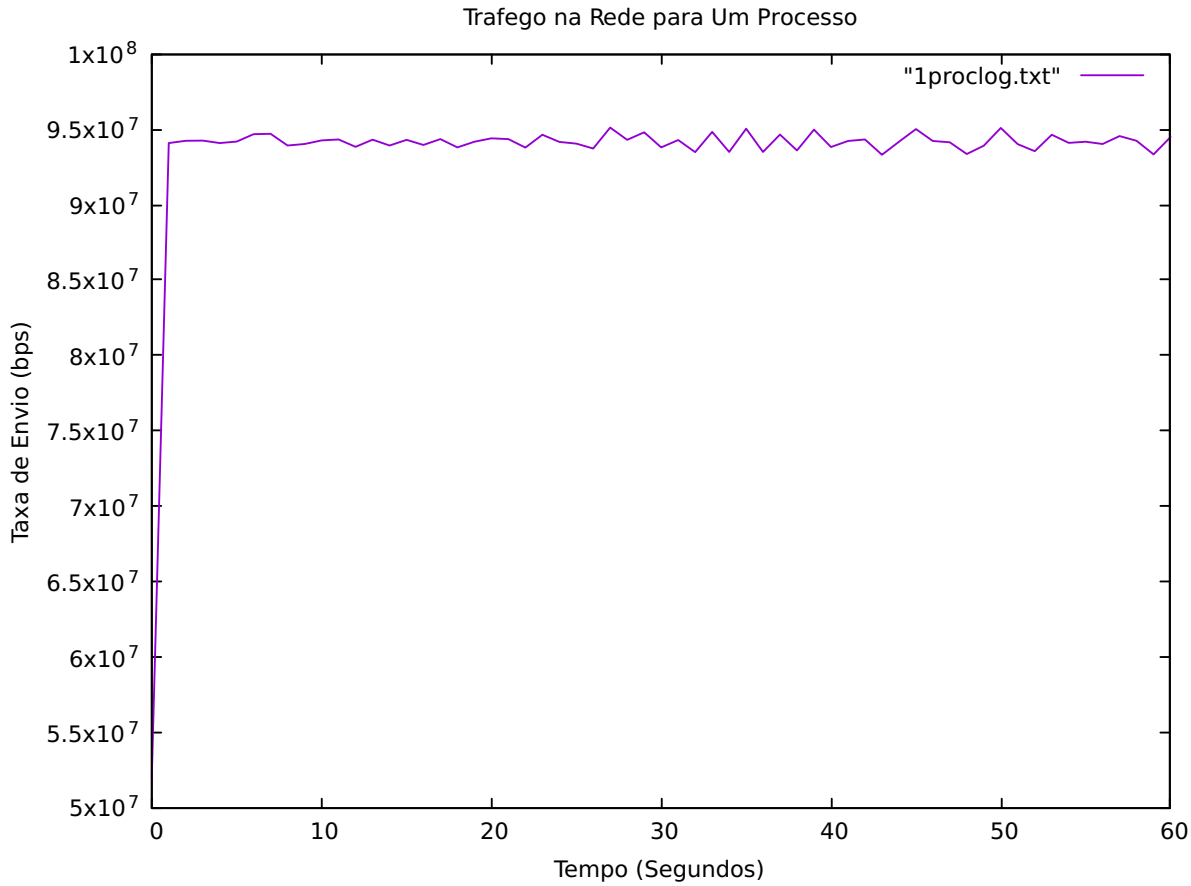

Figura 11 – Gráfico De Tráfego Com Um Processo

rede.

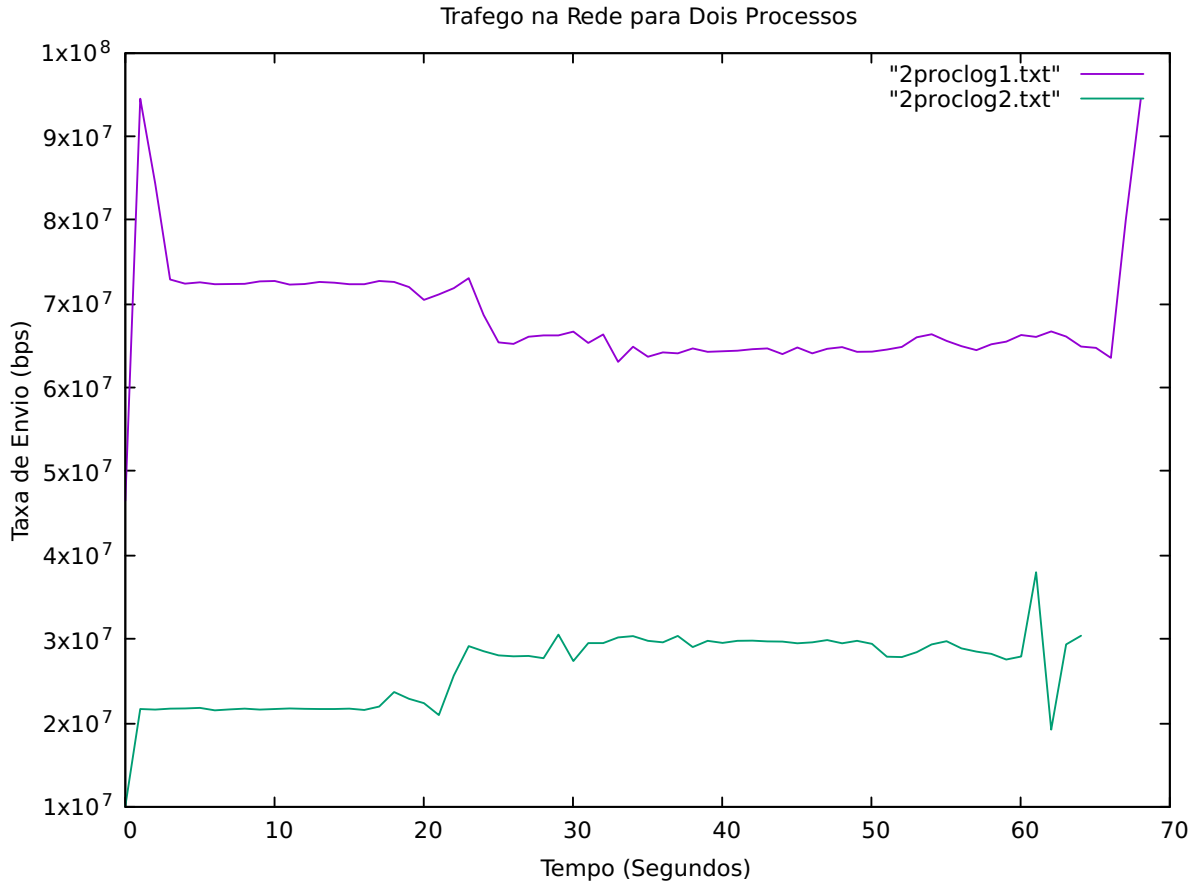

Figura 12 – Gráfico De Tráfego Com Dois Processos

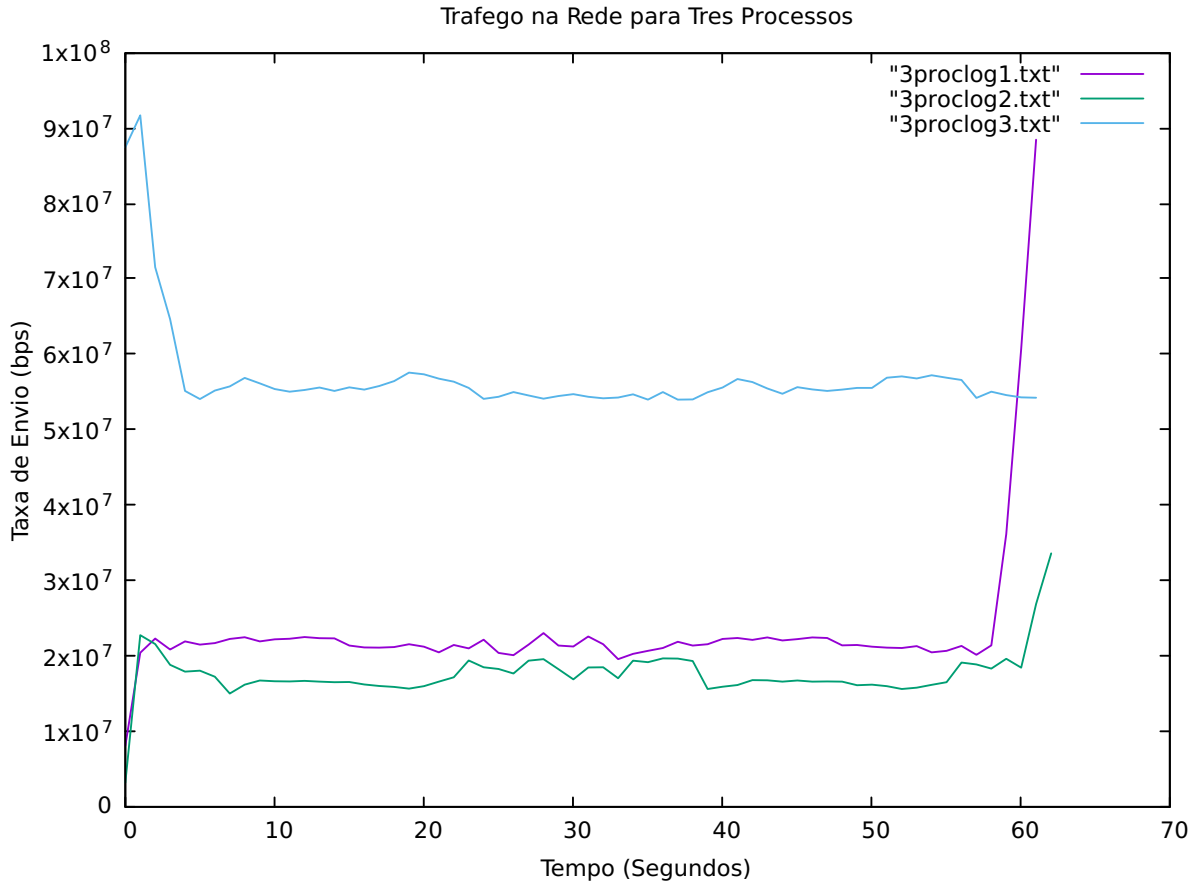

Figura 13 – Gráfico De Tráfego Com Três Processos

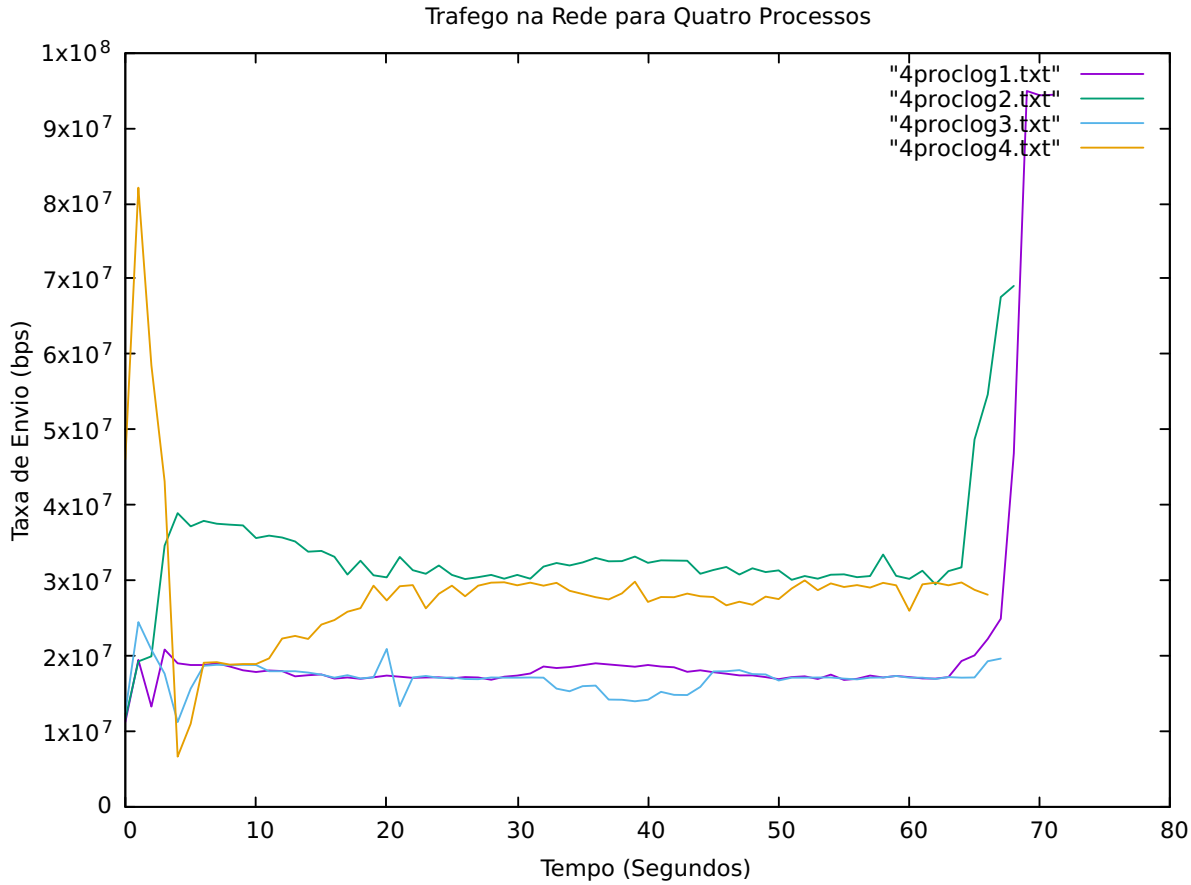

Figura 14 – Gráfico De Tráfego Com Quatro Processos