



ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA
DISCIPLINA: REDES TCP/IP
1º SEMESTRE DE 2010

TRABALHO PRÁTICO 2

Alunos: Alex Sander Ferreira Eusébio
Marco Antônio da Silva Barbosa
Marlon Paolo Lima

INTRODUÇÃO

O presente trabalho detalha a implementação de um servidor para transferência de arquivos baseado no protocolo TFTP, utilizando segmentos UDP. Na aplicação desenvolvida, um cliente recebe dados de um servidor utilizando a porta 51975, que envia blocos de mensagens com tamanhos fixos de 257 bytes, sendo que 256 são pacotes de dados e 1 byte que sinaliza a posição da janela. Um bloco enviado com um tamanho menor que este, significa o término da transferência. O algoritmo desenvolvido deve ser capaz também de simular perdas de pacotes na rede, onde as taxas podem variar entre 0 a 100%. Para uma melhor fundamentação teórica, o grupo realizou um profundo estudo na RFC 1350 abordando aspectos relativos ao processo de comunicação e tratamento de erros. Ao final deste relatório é apresentado o código fonte do algoritmo com as devidas adaptações sugeridas. A fim de avaliar a qualidade do algoritmo produzido, são realizados diversos testes que comprovaram a eficiência do algoritmo.

A RFC1350

Conforme estudos anteriores, as RFCs (Request For Comments) são documentos que especificam e ajudam a debater a maioria das normas e padrões de tecnologias ligados à Internet e às redes em geral. A RFC 1350 especifica o protocolo TFTP (Trivial File Transfer Protocol), um protocolo simples para transferência de arquivo executado sobre protocolo de transporte UDP, onde o cliente é capaz de ler e gravar arquivos em um servidor remoto. O funcionamento deste protocolo é iniciado com um pedido de leitura ou escrita de arquivos, o qual também serve para abrir uma sessão. No caso da leitura, o cliente envia um pedido denominado RRQ (Read Request) contendo a indicação, nome do arquivo e o modo da transferência. Se o servidor reconhecer o pedido, a sessão é aberta e o arquivo é enviado num bloco de tamanho fixo de 512 bytes. Cada pacote de dados contém um bloco de dados e deve ser reconhecido por um pacote de ACK, enviado pelo cliente, antes que o servidor envie um próximo. Se durante o processo um pacote se perde na rede, o timeout acionado no envio do pacote irá expirar, e neste instante o emissor reenviará os dados novamente. Este é o mesmo princípio da transmissão confiável, baseada na solicitação automática de repetição (ARQ), utilizado na técnica Stop-and-wait. Um bloco menor do que o bloco padrão (512 bytes) vai indicar o fim da comunicação.

Erros são previstos no processo de transmissão do TFTP. Na ocorrência destes, uma mensagem de erro é gerada e enviada. Erros comuns neste tipo de comunicação são perdas de pacotes na rede, arquivos não encontrados, recebimento de pacote duplicado e problemas referente à permissões de acesso. Desta maneira ACKs, timeouts e pacotes com mensagens de erro são fundamentais para este tratamento.

CASOS DE USO E PARTICULARIDADES DO TFTP

O TFTP possui várias particularidades em relação ao FTP, porém seu objetivo é bem específico: simplificar a implementação. Daí surge a principal motivação para o uso do protocolo UDP na camada de transporte. Ao contrário do TCP, o UDP é um protocolo simples e leve, que não garante a entrega confiável dos pacotes. Desta forma, fica a critério da aplicação o desenvolvimento de mecanismos para se obter uma transmissão confiável. Outra característica inerente ao FTP é que este utiliza um protocolo capaz de reordenar pacotes que podem chegar fora de ordem, enquanto que o UDP realiza tal tarefa. Novamente deve ser implementado na aplicação algum mecanismo que trate este problema, como o Stop-and-wait, utilizado pelo TFTP.

Assim o protocolo UDP se torna mais adequado à aplicação TFTP, por ser mais simples e necessitar de menos processamento computacional, tornando-o eficiente para aplicações com limitado poder computacional ou em sistemas onde uma transmissão de dados rápida é esperada. Devemos salientar também que segurança não é um ponto forte do TFTP. Este protocolo não provê nenhum mecanismo para autenticação, criptografia de dados.

Um dos casos onde o protocolo TFTP é muito utilizado, é em dispositivos que efetuam boot através da rede. Esse mecanismo permite carregar sistemas operacionais e programas a partir de um servidor na rede. Como para uma rede o ideal é que o fluxo de dados corra de maneira simples e rápida, o TFTP se enquadra perfeitamente.

ORGANIZAÇÃO DO CÓDIGO E ALTERAÇÕES

A codificação dos programas cliente e servidor foram feitas a partir dos programas do TP1 que basicamente encaminhavam um buffer via UDP entre dois hosts. Para facilitar o entedimento e estruturação do programa, foram criadas funções para tratar procedimentos de forma individual diferente do primeiro trabalho prático onde tudo estava apenas na função `main()`. Os procedimentos implementados no lado servidor com respectivas funções foram:

Destrava: função acionada pelo signal para destravar o flag do loop de aguardo do ACK do cliente, representando o estouro de timeout;

AbreArquivo: função acionada na função `main`, assim que recebido um pedido do cliente, para abrir o arquivo solicitado para leitura, retornando ao cliente o sucesso ou insucesso na abertura deste arquivo, este retorno é feito na forma de uma Mensagem apontando “Arquivo aberto com sucesso.” ou “Erro na abertura do arquivo!”. Neste procedimento é impresso no `stdout` do servidor as mensagens que apontam sucesso ou insucesso na abertura do arquivo.

PegaBloco: função que recupera 256 bytes do arquivo solicitado, ou menos caso o arquivo termine antes do bloco inteiro ser preenchido, a função retorna 1 quando fim de arquivo ou 0 em caso contrário.

MontaPacote: monta um vetor de caracteres com 257 bytes, onde no primeiro byte é colocado 0 ou 1 dependendo do valor da janela de transmissão e nos bytes seguintes é concatenado o bloco recuperado do arquivo solicitado pelo cliente.

AguardaACK: função mais importante do servidor, onde inicialmente é ativado o signal para rodar a função `Destrava` caso o alarm extrapole o tempo de 3 segundos (tempo estipulado aleatoriamente). Após ativado a interrupção um loop é iniciado, este loop executa a função `recvfrom` do UDP, mas com um parâmetro que indica à função para não entrar em estado de espera até o pacote chegar do cliente, isto provoca uma execução contínua do loop testando a cada momento se um pacote chegou. Se o pacote chegar antes do alarm estourar, o alarm é zerado e o flag do loop destravado, caso contrário o alarm dispara, executando o signal que aciona a função `Destrava` que força a finalização do loop. Se o um pacote chegar ele é testado para verificar se é uma confirmação (ACK) o que encerra a função retornando o valor 1 e desativando o alarm, caso contrário pode ser um pedido de nova conexão que será negada e neste caso o alarm não é interrompido e o processamento continua aguardando a confirmação ou estouro do timeout. O cliente que

solicitou um novo arquivo é informado que o servidor está ocupado e no prompt do servidor é impresso um aviso de tentativa de novo download.

EnviaPacote: função que encaminha o pacote com 257 bytes, ou menos se fim de arquivo, para o cliente e imprime na saída padrão o envio, indicando qual a janela de transmissão do pacote.

Main: função principal da aplicação que declara a maioria das variáveis, inicializa o socket e o coloca em modo de espera. Esta função possui três loops que coordenam o comportamento do servidor. Um que mantém o servidor sempre em espera por um novo pedido de arquivo, ou seja, apenas atendida por completo uma solicitação o servidor entra em modo de espera por um novo pedido. O segundo loop é feito para mandar os pacotes com os dados do arquivo para o cliente até que o arquivo chegue ao seu fim. O terceiro loop aguarda a confirmação de entrega do pacote para autorizar o envio do próximo. Em linhas gerais algo parecido com o algoritmo resumido abaixo:

```
Enquanto verdadeiro faça
    Aguarde novo cliente
    Abra o arquivo solicitado
    Enquanto não terminar o arquivo solicitado faça
        Recupere bloco do arquivo
        Monta pacote
        Enquanto não receber confirmação
            Envia o pacote
            Aguarda confirmação ou timeout
        Fim
    Fim
Fim
```

O código do cliente também foi segmentado em funções para facilitar a programação, manutenção e entedimento. As funções criadas foram as seguintes:

Destrava: assim como no servidor, usada para alterar o valor do flag de loop no aguardo de recebimento de pacotes vindas do servidor, neste caso, para evitar que o cliente trave aguardando indefinidamente a chegada de um pacote do servidor, seja este pacote indicando sucesso ou insucesso na abertura do arquivo ou para entregar um bloco de dados do arquivo solicitado.

PedeAberturaArquivo: encaminha para o servidor o nome do arquivo solicitado pelo usuário, em seguida ativa o signal com tempo de espera de 10 segundos (tempo definido aleatoriamente) e entra em um loop aguardando a resposta do servidor. Se a resposta chegar ela é verificada para identificar sucesso ou insucesso na abertura. Em caso de sucesso o programa continua operação, em caso de insucesso o programa é interrompido, mesmo procedimento caso a resposta não retorne do servidor após 10 segundos.

RecebePacote: função que recebe pacote de dados do servidor, um signal de 10 segundos é disparado para aguardar o pacote do servidor, este procedimento é necessário para evitar que cliente permaneça travado indefinidamente. Em caso de recebimento correto a string pacote é carregada com os dados recebidos do cliente. Lembrando que no primeiro byte destes dados está a informação de qual janela aquele pacote pertence e nos demais bytes os caracteres recuperados do arquivo solicitado. Esta função retorna a quantidade de bytes recebidos do servidor o que vai permitir concluir se o a transmissão do arquivo terminou ou não.

GravaBloco: escreve em arquivo os bytes relativos ao arquivo solicitado, sem colocar o byte que indica a posição da janela do transmissor. Retorna o valor da próxima janela aguardada.

Envia ACK: função que retorna ao servidor um ACK para cada pacote recebido com dados de arquivo. Em particular esta função implementa perda de recebimento e envio de acordo com parâmetro de entrada do cliente. Logo, esta função ao ser chamada calcula randômicamente o erro, se o valor randômico for menor que erro, a função considera que houve perda no recebimento do pacote e apenas imprime o erro no prompt e retorna 0 como forma de indicar que o ACK não foi enviado. Se não for gerado o falso erro no recebimento, a função imprime que não houve erro no recebimento e calcula novamente uma percentagem de erro para envio do ACK. Considerando que não vai ser gerado erro de envio do ACK, um ACK será enviado ao servidor e uma mensagem indicando este envio é impressa no terminal. Caso tenha sido gerado um erro de envio de ACK a função apenas

imprime na tela a falha. Esta função retorna 0 em caso de insucesso de envio do ACK ou 1 em caso de sucesso.

Main(): função principal que primeiro testa o número de argumentos digitados pelo usuário e o valor limite da taxa de erro desejada. Em seguida é verificado também o hostname digitado e tendo sucesso nestas verificações o socket é criado e vinculado a porta definida. É apresentada mensagem no prompt confirmando ao usuário o nome do arquivo solicitado e a taxa de erro pretendida. No próximo passo o cliente encaminha para o servidor o pedido de transferência do arquivo, em caso de sucesso na abertura por parte do servidor o cliente abre o arquivo “arquivo_recebido.txt” para escrever os bytes que serão recebidos. A função random é inicializada e o programa é colocado em loop enquanto não for verificado o fim do arquivo solicitado. Outro loop aguarda o recebimento do pacote (RecebePacote), grava em disco se realmente for o pacote esperado (GravaBloco) e tenta mandar ACK (EnviaACK), este segundo loop só é interrompido se for realmente encaminhado um ACK. Antes de escrever em arquivo, a função verifica se o pacote recebido é realmente aquele da janela esperada, pois pode ter acontecido um erro real de transmissão do ACK e o servidor pode reencaminhar um pedaço do arquivo já escrito em disco.

O cliente interrompe sua execução se permanecer mais de 10 segundos sem receber dados do servidor. Em contrapartida o servidor depois de 16 tentativas de transmissão também considera comunicação perdida e entra em estado de espera por novo cliente.

O EXPERIMENTO

O desenvolvimento de um servidor TFTP solicitado no Trabalho prático 2 difere-se da proposta de implementação da RFC 1350 em alguns aspectos. Como mencionado na descrição do trabalho, este servidor será inspirado na norma publicada pelo IETF. Isto implica que algumas das características e funcionalidades propostas na RFC 1350 não farão parte da aplicação desenvolvida neste trabalho prático.

Na proposta da RFC 1350, é possível um cliente ler e gravar arquivos em um servidor TFTP remoto [1]. Porém, neste trabalho prático, será desenvolvida uma aplicação cliente-servidor "read-only", ou seja, o cliente só poderá ler os arquivos que estarão no

servidor. Desta maneira, torna-se desnecessária a inclusão no trabalho da função WRQ (Write Request) descrita na RFC.

Outra diferença entre a implementação solicitada no trabalho e a sugerida pela norma, diz respeito ao tamanho do bloco de dados enviados. A RFC propõe que cada bloco de dados terá um tamanho mínimo de 512 bytes. Menos que isto, o protocolo considera que este será o último pacote a ser transmitido. O TP 2 indica que cada bloco terá 256 bytes (a metade). O princípio para término da transmissão segue o mesmo proposto pela RFC, ou seja, pacotes menores que 256 bytes (tamanho especificado) encerra a transmissão, após o envio e recepção do ACK correspondente.

Uma importante funcionalidade descrita na RFC 1350, que será utilizado neste trabalho, é o esquema de confirmação no recebimento e retransmissão de pacotes. Apesar de o servidor enviar pacotes de dados para o cliente através de segmentos UDP, serão implementadas algumas funcionalidades na aplicação, inspiradas no protocolo TCP. É preciso deixar bem claro que este serviço (transferência de arquivos através do TFTP) foi concebido para ser bem simples e leve, por isto, neste caso o protocolo UDP é mais indicado para transferência de dados. Por último, a porta UDP utilizada para a comunicação sugerida pela norma deve estar entre 0 a 65.535, mas neste caso específico será utilizada porta 51.975.

TESTES

O grupo realizou os seguintes testes na etapa de experimentos:

- testes de conectividade e leitura de arquivos;
- tentativa de conexão simultânea de dois clientes TFTP com um mesmo servidor;
- simulação de uma queda no link de dados, onde a transmissão é interrompida;
- teste de envio de arquivo na rede, com taxas de perda variando entre 0 a 60%;
- Tentativa de acesso de um arquivo inexistente;
- transmissão de um arquivo grande (mais de 2 MB);
- comparativo de desempenho x taxa de erro;

ANÁLISE DOS RESULTADOS

No primeiro teste realizado, o servidor é habilitado na forma passiva para atender uma requisição de leitura de um pequeno arquivo (10 KB) pelo cliente com taxa de perda de 0%. No experimento, o cliente realiza a solicitação informando o endereço do servidor, nome do arquivo e a taxa de perda. Percebe-se que o protocolo TFTP atua com sucesso, como pode ser visto na figura 1 que mostra o processo de transferência no cliente e no servidor.

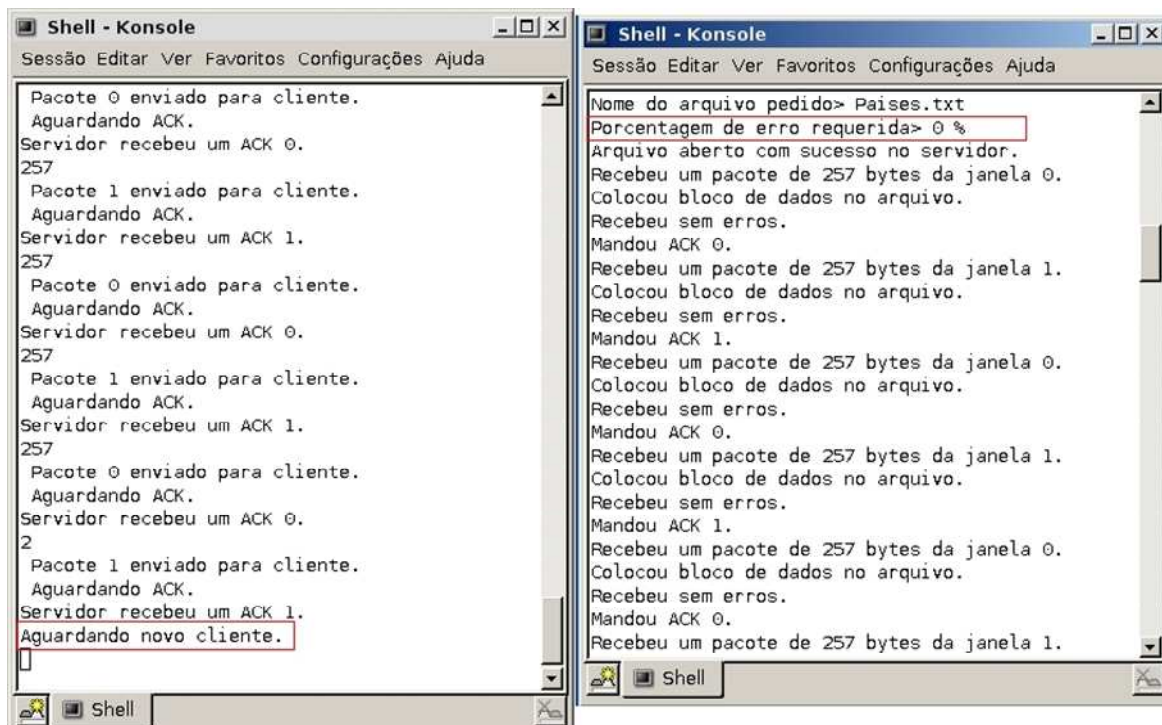


Figura 1: Transferência de arquivo sem taxa de perda.

No segundo, foi realizada a transferência do mesmo arquivo de texto a uma taxa de erro de 10%. Percebe-se que em ambos testes, o arquivo a ser enviado possui um tamanho superior ao tamanho do pacote usado no TFTP. Assim para o envio dos dados, os pacotes são fragmentados em blocos de 257 bytes (256 bytes para dados e 1byte para sinalizar a posição da janela). Devido à presença de erros, houve um atraso considerado no envio (em média, 31 segundos). Detalhes do envio podem ser visto na figura2.

```
Shell - Konsole
Sessão Editar Ver Favoritos Configurações Ajuda
Pacote 0 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 0.
257
Pacote 1 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 1.
257
Pacote 0 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 0.
257
Pacote 1 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 1.
257
Pacote 0 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 0.
257
Pacote 1 enviado para cliente.
Aguardando ACK.
Estourou o timer, encaminha pacote de novo.
257
Pacote 1 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 1.

Shell - Konsole
Sessão Editar Ver Favoritos Configurações Ajuda
sh-3.2$ ./cliente 10.0.10.23 Países.txt 10
Nome do arquivo pedido> Países.txt
Porcentagem de erro requerida> 10 %
Arquivo aberto com sucesso no servidor.
Recebeu um pacote de 257 bytes da janela 0.
Colocou bloco de dados no arquivo.
Forçou erro no recebimento.
Recebeu um pacote de 257 bytes da janela 0.
perdeu ACK na rede.
Recebeu sem erros.
Mandou ACK 0.
Recebeu um pacote de 257 bytes da janela 1.
Colocou bloco de dados no arquivo.
Recebeu sem erros.
Forçou erro de envio de ACK.
Recebeu um pacote de 257 bytes da janela 1.
perdeu ACK na rede.
Recebeu sem erros.
Forçou erro de envio de ACK.
Recebeu um pacote de 257 bytes da janela 1.
perdeu ACK na rede.
Recebeu sem erros.
Mandou ACK 1.
Recebeu um pacote de 257 bytes da janela 0.
Colocou bloco de dados no arquivo.
Recebeu sem erros.
Mandou ACK 0.
```

Figura 2: Envio de arquivo a taxa de perda de 10 %

Continuando os experimentos, o próximo consistia em fazer um pedido de um arquivo a uma taxa de erro 30%. Apesar da demora (2 minutos e 43 segundos), a transmissão foi realizada sem maiores problemas.

```
Shell - Konsole
Sessão Editar Ver Favoritos Configurações Ajuda
Aguardando novo cliente.
Recebeu um pedido do arquivo Países.txt.
Arquivo aberto com sucesso.
Mensagem de sucesso encaminhada para o cliente.
257
Pacote 0 enviado para cliente.
Aguardando ACK.
Estourou o timer, encaminha pacote de novo.
257
Pacote 0 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 0.
257
Pacote 1 enviado para cliente.
Aguardando ACK.
Estourou o timer, encaminha pacote de novo.
257
Pacote 1 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 1.
257
Pacote 0 enviado para cliente.
Aguardando ACK.
Estourou o timer, encaminha pacote de novo.
257
Pacote 0 enviado para cliente.
Aguardando ACK.
Estourou o timer, encaminha pacote de novo.
257

Shell - Konsole
Sessão Editar Ver Favoritos Configurações Ajuda
sh-3.2$ ./cliente 10.0.10.23 Países.txt 30
Nome do arquivo pedido> Países.txt
Porcentagem de erro requerida> 30 %
Arquivo aberto com sucesso no servidor.
Recebeu um pacote de 257 bytes da janela 0.
Colocou bloco de dados no arquivo.
Forçou erro no recebimento.
Recebeu um pacote de 257 bytes da janela 0.
perdeu ACK na rede.
Recebeu sem erros.
Mandou ACK 0.
Recebeu um pacote de 257 bytes da janela 1.
Colocou bloco de dados no arquivo.
Forçou erro no recebimento.
Recebeu um pacote de 257 bytes da janela 1.
perdeu ACK na rede.
Recebeu sem erros.
Mandou ACK 1.
Recebeu um pacote de 257 bytes da janela 0.
Colocou bloco de dados no arquivo.
Forçou erro no recebimento.
Recebeu um pacote de 257 bytes da janela 0.
perdeu ACK na rede.
Forçou erro no recebimento.
Recebeu um pacote de 257 bytes da janela 0.
perdeu ACK na rede.
Recebeu sem erros.
Forçou erro de envio de ACK.
```

Figura 3: Transferência de um arquivo com taxa de perda de 30%

Porém, ao fixar uma taxa de erro de 60% no envio do arquivo iptables.txt, ocorreu um grande problema. A todo o momento era necessário o reenvio de um pacote, e isto gerou um enorme atraso na transmissão, conforme figura 4:

The image shows two terminal windows side-by-side, both titled 'Shell - Konsole'. The left window displays a log of network communication between a server and a client. The right window shows the client's perspective, including the command used to start the transfer and the received data blocks. Red annotations highlight specific parts of the logs.

Left Window Log:

```
257
Pacote 0 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 0.
257
Pacote 1 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 1.
257
Pacote 0 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 0.
257
Pacote 1 enviado para cliente.
Aguardando ACK.
Estourou o timer, encaminha pacote de novo.
257
Pacote 1 enviado para cliente.
Aguardando ACK.
Servidor recebeu um ACK 1.
257
Pacote 0 enviado para cliente.
Aguardando ACK.
Estourou o timer, encaminha pacote de novo.
257
Pacote 0 enviado para cliente.
Aguardando ACK.
Estourou o timer, encaminha pacote de novo.
257
Pacote 0 enviado para cliente.
Aguardando ACK.
```

Right Window Log:

```
sh-3.2$ ./cliente 10.0.10.23 iptables.txt 60
Nome do arquivo pedido iptables.txt
Porcentagem de erro requerida > 60 %
Arquivo aberto com sucesso no servidor.
Recebeu um pacote de 257 bytes da janela 0.
Colocou bloco de dados no arquivo.
Recebeu sem erros.
Mandou ACK 0.
Recebeu um pacote de 257 bytes da janela 1.
Colocou bloco de dados no arquivo.
Forçou erro no recebimento.
Recebeu um pacote de 257 bytes da janela 1.
perdeu ACK na rede.
Forçou erro no recebimento.
Recebeu um pacote de 257 bytes da janela 1.
perdeu ACK na rede.
Recebeu sem erros.
Mandou ACK 1.
Recebeu um pacote de 257 bytes da janela 0.
Colocou bloco de dados no arquivo.
Forçou erro no recebimento.
Recebeu um pacote de 257 bytes da janela 0.
perdeu ACK na rede.
Forçou erro no recebimento.
Recebeu um pacote de 257 bytes da janela 0.
perdeu ACK na rede.
Forçou erro no recebimento.
Recebeu um pacote de 257 bytes da janela 0.
perdeu ACK na rede.
Forçou erro no recebimento.
```

Figura 4: Transferência de um arquivo com taxa de perda de 60%

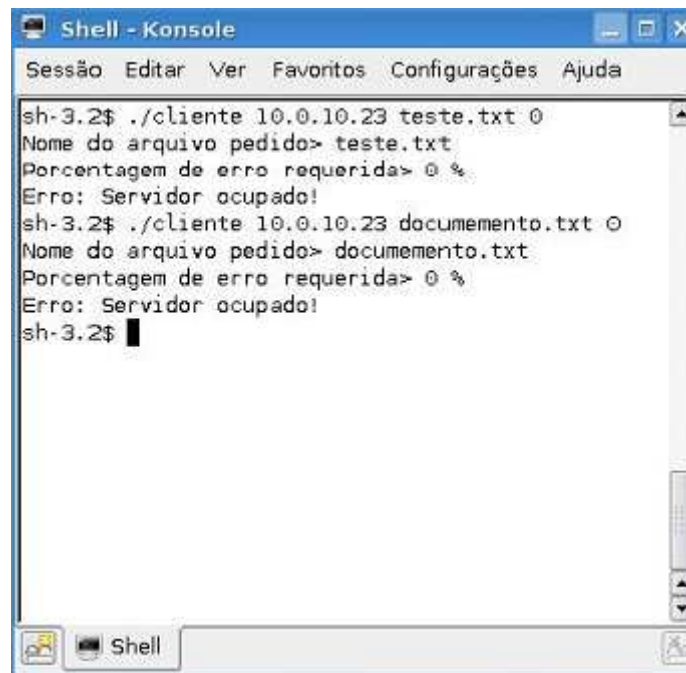
Percebe-se que devido à elevada taxa de perdas, o tempo de envio subiu demasiadamente. Interrompemos os testes quando se passaram 8 minutos e a transferência ainda não havia sido concluída. Devemos levar em consideração, que um erro pode ser gerado tanto no envio do pacote, quanto no envio do ACK, e em ambos casos, haverá a necessidade de retransmissão dos dados. Isto quer dizer que, dos ACKs enviados pelo cliente, haverá também 60% de perdas, elevando o tempo total para retransmissão. Além disso, ainda existe a possibilidade de uma retransmissão falhar novamente, o que implica em perda de ACK e novo timeout, o que deixa o cenário ainda pior.

Uma possível melhoria neste esquema seria reduzir o tempo do timeout de 3 segundos implementado na aplicação para 1 segundo. Se considerarmos a afirmação que Peterson [2] descreve em seu livro, que um RTT de 100 ms é suficiente para atravessar os EUA, podemos concluir que um timeout de 1 segundo atende à maioria das aplicações, mesmo a longas distâncias.

Com esta alteração no algoritmo, percebemos uma melhora considerável no tempo total gasto pelos timeouts para enviar o arquivo. Isto significa um enorme ganho e torna esta aplicação muito mais eficiente. Com a alteração, o tempo médio para transmissão caiu para 3 minutos e 42 segundos.

Adicionalmente, foram realizados experimentos no envio de um arquivo maior (2 MB) para o cliente TFTP. Nestes experimentos foram realizadas simulações com taxa de perda de 0 e 30%. O teste sem perdas foi realizado com êxito. O servidor TFTP enviou rapidamente (em média, 3 segundos) o arquivo para o cliente. No experimento onde foi fixado uma taxa de erro de 30%, surgiram problemas. Houve uma demora excessiva, e interrompemos os testes quando se passaram 10 minutos. Então, optamos por realizar algumas contas e verificamos que para enviar um arquivo de 2 MB, fragmentados em pacotes de 256 bytes, seriam enviados aproximadamente 8.192 pacotes. Se considerarmos uma perda de 30% dos pacotes, quer dizer que mais de 2.400 pacotes seriam considerados com perdido, isto só de recebimento e os que fossem considerados recebidos ainda teriam que passar pelo erro de 30% no envio de ACK. Mesmo com a nova implementação, onde um ACK não recebido é disparado um timeout de 1 segundo o tempo projetado para o experimento ficaria em torno de 40 minutos. Realizamos mais alguns testes com uma taxa de perda de 60%. Percebe-se na prática que, transferir um arquivo, mesmo que pequeno (200 KB) em uma rede com tantos erros, é praticamente inviável. Neste caso, haverá uma quantidade de retransmissão enorme, e mesmo reduzindo o tempo do timeout para um valor abaixo de 100 ms, os resultados foram péssimos. Devemos levar em consideração que a escolha de um valor de timeout apropriado não é tarefa fácil [2]. Então, reduzir demasiadamente o timeout dos pacotes pode trazer sérios problemas para uma transmissão que utiliza o método Stop-and-wait, principalmente se o RTT da rede for grande. Figuras dos testes podem ser visualizadas abaixo.

Dentre os vários experimentos executados um deles especificamente consistia na solicitação de alguns clientes simultâneos a um mesmo servidor. Ocorre que durante esse processo de execução, o programa aceita apenas um cliente. Os clientes que não conseguirem se conectar, a aplicação encaminha uma mensagem de erro informando que o servidor está ocupado.



```
Shell - Konsole
Sessão Editar Ver Favoritos Configurações Ajuda

sh-3.2$ ./cliente 10.0.10.23 teste.txt 0
Nome do arquivo pedido> teste.txt
Porcentagem de erro requerida> 0 %
Erro: Servidor ocupado!
sh-3.2$ ./cliente 10.0.10.23 documemento.txt 0
Nome do arquivo pedido> documemento.txt
Porcentagem de erro requerida> 0 %
Erro: Servidor ocupado!
sh-3.2$
```

Figura 5: cliente tentando acessar um servidor ocupado.

Um cenário que pode ocorrer em qualquer comunicação é uma queda no link dados, seja por falhas na rede, interferência, dentre outros. Assim, simulamos este cenário interrompendo a comunicação física entre o cliente e servidor. Porém, percebemos um novo problema em nossa aplicação. Quando tal fato ocorre, o servidor fica enviando o mesmo pacote inúmeras vezes para um cliente, que pode até não mais existir, e o sistema fica esperando uma confirmação de ACK. O resultado deste problema seria desastroso em um servidor TFTP na rede. Outros clientes, ou até o mesmo cliente que perdeu sua conexão simplesmente não iria conseguir se comunicar com o servidor, pois ele está esperando um ACK que talvez nunca chegue. Ao realizarmos uma leitura na RFC 1350, não encontramos nenhum aspecto que trata deste problema específico. Assim uma melhoria foi introduzida no servidor, que trata deste problema da seguinte maneira: quando o servidor tenta se comunicar por 16 vezes com o cliente e recebe nenhuma confirmação, automaticamente é encerrada a sessão, exibindo uma mensagem de erro (excedeu 16 tentativas de enviar o mesmo pacote). Na aplicação cliente, foi implementado um alarm que encerra a aplicação caso nenhum pacote chegue do servidor após 10 segundos. O resultado desta implementação pode ser vista na figura 6.

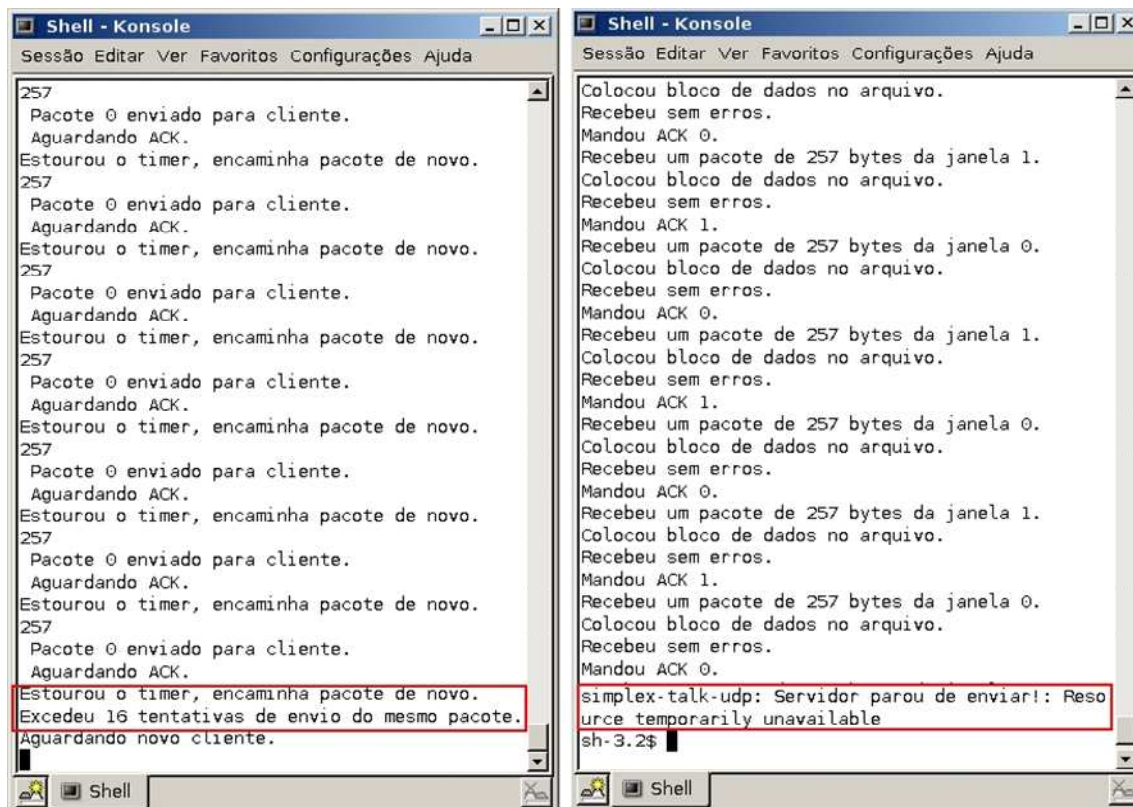


Figura 6: Perda de conexão e encerramento de conexão

Por fim verificamos como o sistema se comportaria com uma requisição de um arquivo inexistente. Nota-se que na implementação do trabalho uma mensagem de erro na abertura do arquivo é gerada no servidor e cliente, conforme pode ser verificada na figura 7.

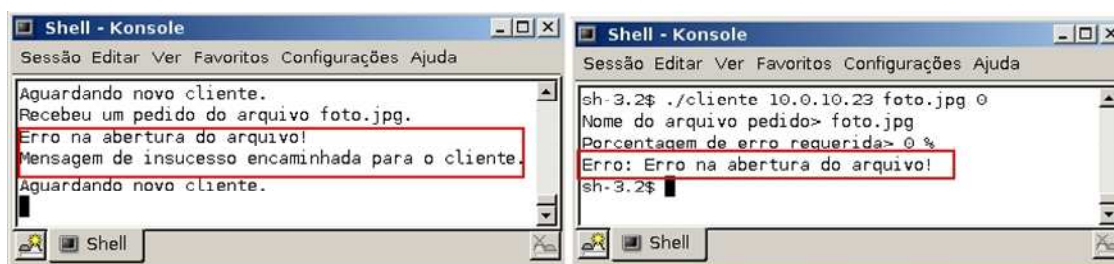


Figura 7: arquivo inexistente

CONCLUSÃO

O trabalho prático 2 desenvolvido na disciplina foi importante para contextualizar e aplicar os princípios da transmissão confiável para o tratamento de erros. A partir de pesquisa bibliográfica sobre o assunto, foi possível o desenvolvimento de uma aplicação

simples e prática para transferência de arquivos entre cliente e servidor, utilizando o protocolo orientado a fluxo UDP. Assim, foi possível comprovar a eficiência e rapidez do protocolo TFTP em ambientes onde a taxa de erro não seja elevada.

Diante dos resultados obtidos neste trabalho, podemos concluir que apesar do protocolo UDP não confirmar o recebimento de mensagens, nem tratar a retransmissão de pacotes, é possível implementar esta funcionalidade na camada de aplicação, o que tornaria um sistema confiável mesmo utilizando um protocolo não confirmado e não orientado à conexão para transferência dos dados. Percebe-se também com a prática que um serviço TFTP não é indicado para transmissão de arquivos grande, principalmente em redes que podem ocorrer falhas.

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] SOLLINS, K., "The TFTP Protocol (Revision 2)", STD 33, RFC 1350, MIT, July 1992.
- [2] PETERSON, Larry L., DAVIE, Bruce S. *Redes de Computadores: Uma abordagem de Sistemas*. 3 ed. São Paulo, ELSEVIER, 2004.
- [3] HALL, Brian B. J. *Beej's Guide to Network Programming: Using Internet Sockets*. Disponível em <http://beej.us/guide/bgnet/output/html/multipage/index.html>, 2009.
- [4] KUROSE, James F., ROSS, Keith W. *Redes de Computadores e a Internet: Uma abordagem top.down*. 3 ed. São Paulo, Addison Wesley, 2005.

ANEXO I - CÓDIGO FONTE COMENTADO

CLIENTE TFTP

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <signal.h>

#define SERVER_PORT 51975
#define MAX_LINE 257

FILE * Arq;
int flag;
char pacote[MAX_LINE+1];

void Destrava ()
{ /* Destrava flag se pacote do servidor nao voltar a tempo */
    flag = 1;
}

void PedeAberturaArquivo ( int socket, struct sockaddr_in servidor, char NomeArquivo[30] ) {
/* encaminha para servidor nome de arquivo a ser aberto e printa o sucesso ou insucesso */
    char p[MAX_LINE + 1];
    int len;
    int tam_sin = sizeof(servidor);

    /* manda nome do arquivo para servidor */
    if (sendto (socket, NomeArquivo, strlen(NomeArquivo) +1, 0, (struct sockaddr *)&servidor, sizeof
(servidor)) < 0 ) {
        perror("simple-tftp: Erro Send to do Nome do Arquivo");
        exit (1);
    };

    signal (SIGALRM, Destrava);
    alarm (10);
    flag = 0;

    while ( ! flag ){
        /* recebe sucesso ou falha do servidor na abertura do arquivo */
        len = recvfrom (socket, p, sizeof(p), MSG_DONTWAIT, (struct sockaddr *)&servidor,
&tam_sin);
        if ( len > 0 ) {
            alarm (0);
            flag = 1;
        };
    };

    if (len < 0){
        perror("simplex-tftp: Servidor nao respondeu!");
        exit (1);
    };

    if ( !strcmp ( p, "Arquivo aberto com sucesso.", 15) ) {
        fprintf ( stdout, "Arquivo aberto com sucesso no servidor.\n");
    } else {
        fprintf (stdout, "Erro: %s\n", p);
        exit (1);
    }
}
```

```

    }
}

int RecebePacote ( int socket, struct sockaddr_in servidor ){

    int tamanho;
    int tam_sin = sizeof (servidor);

    signal (SIGALRM, Destrava);
    flag = 0;
    alarm (10);
    pacote[0] = '\0';
    while (!flag) {
        tamanho = recvfrom (socket, pacote, sizeof(pacote), MSG_DONTWAIT, (struct sockaddr
*)&servidor, &tam_sin);
        if ( tamanho > 0 ) {
            alarm (0);
            flag = 1;
        }
    }

    if (tamanho < 0 ) {
        perror("simplex-talk-udp: Servidor parou de enviar!");
        exit (1);
    }
    fprintf (stdout, "Recebeu um pacote de %d bytes da janela %c.\n", tamanho-1, pacote[0]);
    return tamanho;
}

```

```

char GravaBloco ( int tamanho ){
    /* grava em arquivo o bloco recebido e devolve a proxima janela esperada */

    int i;
    char retorno;

    for ( i=1; i < (tamanho-1); i++ )
        fputc (pacote [i], Arq);

    fprintf ( stdout, "Colocou bloco de dados no arquivo.\n" );

    /* indica qual eh o proximo bloco esperado */

    if ( pacote[0] == '0' ){
        retorno = '1';
    }
    else {
        retorno = '0';
    }
    return retorno;
}

```

```

int EnviaACK (int socket, struct sockaddr_in servidor, int erro, char w) {

    int len;
    int i, j;
    char sACK[30];
    int ACK = 0;

    j = rand()%100;
    if ( j < erro ) {
        /* forca erro no recebimento do pacote */
        /* Nao faz nada com informacao que chegou */
    }
}

```

```

        fprintf ( stdout, "Forçou erro no recebimento.\n");
    }
    else {
        /* pacote chegou sem "erro" */
        fprintf ( stdout, "Recebeu sem erros.\n");

        j = rand () % 100;
        if ( j >= erro ) {
            if ( w == '0')
                strcpy (sACK , "ACK 0");
            else
                strcpy (sACK, "ACK 1");

            if ( sendto (socket, sACK, strlen(sACK) +1, 0, (struct sockaddr *)&servidor,
sizeof (servidor) ) < 0 ) {
                perror ("simplex-tftp: Erro ACK.");
                exit (1);
            };
            ACK = 1;
            fprintf (stdout, "Mandou %s.\n", sACK);
        }
        else {
            fprintf (stdout, "Forçou erro de envio de ACK.\n");
        }
    }
    return ACK;
}

int main (int argc, char * argv[]) {
    FILE *fp;
    struct hostent *hp;
    struct sockaddr_in sin; /* estrutura das informações do servidor */
    struct sockaddr_in cin; /* estrutura das informações do cliente */
    char *host;
    char bloco[MAX_LINE]; /* variavel com o bloco de dados do arquivo */
    char NomeArq[30]; /* variavel com nome do arquivo passado como parametro de chamada do
cliente */
    int s;
    int erro; /* porcentagem de erro requerida */
    int len;
    int tam_sin; /* recebe o tamanho da estrutura do servidor */
    char Esperado;
    int i; /* para loop de escrita no arquivo */
    int ACK; /* aponta se ACK foi mandado ou nao */

    /*confere se os tres parametros foram digitados */
    if (argc == 4) {
        host = argv[1];
        strcpy (NomeArq, argv[2]);
        erro = atoi (argv[3]);
    } else {
        fputs ("Quantidade de paramentos incorreta, digite: ", stdout );
        fputs ("cliarq nome_host nome_arq <0-100>", stdout);
    }

    /* testa para ver se erro digitado pelo usuario estah entre 0 e 100 */
    if (erro <0 || erro > 100) {
        fprintf (stdout, "Erro deve estar entre 0 e 100.\n");
        exit (1);
    }
}

```

```

/* traduz nome do host para endereço IP do par */
hp = gethostbyname (host);
if (!hp) {
    fprintf (stderr, "simple-tftp: host desconhecido: %s\n", host );
    exit (1);
}

```

```

/* monta estrutura de dados do endereço */
bzero ((char *) &sin, sizeof (sin));
sin.sin_family = AF_INET;
bcopy(hp->h_addr, (char*)&sin.sin_addr, hp->h_length);
sin.sin_port = htons (SERVER_PORT);

```

```

cin.sin_family = AF_INET;
cin.sin_port = htons(0);
cin.sin_addr.s_addr= htonl(INADDR_ANY);

```

```

/* abertura ativa */
if ((s= socket (AF_INET, SOCK_DGRAM, 0)) < 0 ) {
    perror ("simplex-talk-udp: socket");
    exit (1);
}

```

```

if (bind (s, (struct sockaddr*) &cin, sizeof(cin)) <0){
    perror ("simplex-talk-udp: bind");
    exit (1);
};

```

```

/* texto para melhorar interface com usuário */
fprintf ( stdout, "Nome do arquivo pedido> %s \n", NomeArq );
fprintf ( stdout, "Porcentagem de erro requerida> %d %\n", erro );

```

```

PedeAberturaArquivo (s, sin, NomeArq);

```

```

/* abre arquivo para escrita */
Arq = fopen ("arquivo_recebido.txt", "w");

```

```

/* o primeiro pacote esperado eh da janela 0 */
Esperado = '0';

```

```

/* incializando a semente de aleatoriedade */
srand ( time (NULL) );

```

```

do { /* enquanto nao for fim de arquivo faca */
    ACK = 0;
    do { /* enquanto nao enviar o ack */
        pacote[0] = '\0';
        len = RecebePacote ( s, sin );
        if ( pacote[0] == Esperado )
            Esperado = GravaBloco ( len );
        else
            fprintf ( stdout, "perdeu ACK na rede.\n");

        ACK = EnviaACK ( s, sin, erro, pacote[0] );
    } while ( ! ACK );
} while ( len == 258 );

```

```

fclose (Arq);

```

```

}

```

SERVIDOR TFTP

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <signal.h>

#define SERVER_PORT 51975
#define MAX_LINE 257
int flag;
FILE * Arq;

void Destrava()
{ /* Detrava flag se pacote nao chegar a tempo */
    flag = 1;
}

int AbreArquivo ( int socket, struct sockaddr_in cliente, char NomeArquivo[30] )
{ /* Abre arquivo solicitado para leitura, indicando sucesso ou insucesso para o cliente */
    char Mensagem [MAX_LINE];
    int tam_cin = sizeof (cliente);
    Mensagem[0] = '\0';
    Arq = fopen (NomeArquivo, "r" );
    if (Arq == NULL) { /* Erro an abertura do arquivo */
        fprintf ( stdout, "Erro na abertura do arquivo!\n" );
        strcpy ( Mensagem, "Erro na abertura do arquivo!");
        if ( sendto (socket, Mensagem, strlen (Mensagem) + 1, 0, (struct sockaddr *) &cliente, tam_cin) < 0)
        {
            fprintf ( stdout, "Erro no envio de Mensagem para cliente.\n" );
            exit (1);
        }
        fprintf (stdout, "Mensagem de insucesso encaminhada para o cliente.\n");
        return 1;
    }
    else {
        fprintf ( stdout, "Arquivo aberto com sucesso.\n" );
        strcpy ( Mensagem, "Arquivo aberto com sucesso.");
        if ( sendto (socket, Mensagem, strlen (Mensagem) + 1, 0, (struct sockaddr *) &cliente, tam_cin) < 0)
        {
            fprintf ( stdout, "Erro no envio de Mensagem para cliente.\n");
            exit (1);
        };
        fprintf (stdout, "Mensagem de sucesso encaminhada para o cliente.\n");
        return 0;
    };
};

int PegaBloco ( char bloco[MAX_LINE] )
{ /* recupera um bloco de até 256 bytes do arquivo e retorna 0 se não for final do arquivo ou 1 se for final */
    int i = 0;
    do {
        bloco[i] = fgetc (Arq);
        i ++;
    } while ( (i<256) && (bloco[i-1] != EOF) );

    if ( bloco[i-1] == EOF )
        bloco [i-1] = '\0';
    else
        bloco[i]='\0';
    if ( strlen(bloco) < 256 )
```

```

        return 1; /* fim de arquivo */
    else
        return 0;
};

void MontaPacote ( int * W, char pacote[MAX_LINE+1], char bloco[MAX_LINE] )
{ /* monta o pacote a ser transmitido colocando no primeiro byte o número da janela e a partir do 1o byte o
   bloco recuperado do arquivo */
    pacote[0] = '\0';
    if (*W == 0) {
        strcpy ( pacote, "0");
        strcat (pacote, bloco);
        *W = 1;
    } else {
        strcpy ( pacote, "1");
        strcat (pacote, bloco);
        *W = 0;
    }
};

int AguardaACK ( int socket, struct sockaddr_in cliente )
{ /* Dispara um signal alarm por 3 segundos, destravando o loop em caso de timeout, ou interrompe caso
   chegue a confirmação. Devolve 0 se estourou ACK ou 1 se tiver recebido o ACK */
    int tamanho;
    char sACK[30];
    char Mensagem[MAX_LINE];
    int tam_cin = sizeof (cliente);

    /* ativa alarme para aguardar por um tempo determinado o ACK */
    signal (SIGALRM, Destrava);
    flag = 0;
    alarm(1);
    sACK[0] = '\0';
    Mensagem[0] = '\0';

    fprintf ( stdout, "Aguardando ACK.\n");

    while ( !flag ) {
        /* recebe confirmacao ou novo pedido de abertura*/
        tamanho = recvfrom(socket, sACK, sizeof(sACK), MSG_DONTWAIT, (struct sockaddr *)
        &cliente, &tam_cin);
        if (tamanho > 0 ) { /* recebeu pacote */
            /* verifica se eh novo pedido */
            if (!strncmp (sACK, "ACK ", 4)) { /* eh um ACK */
                /* destaiva signal */
                alarm (0);
                fprintf ( stdout, "Servidor recebeu um %s.\n", sACK);
                flag = 1;
                return 1;
            }
            else { /* eh um novo pedido de arquivo */
                fprintf ( stdout, "Servidor Ocupado!\n");
                strcpy ( Mensagem, "Servidor ocupado!");
                if ( sendto (socket, Mensagem, strlen (Mensagem) + 1, 0,(struct sockaddr *)
                &cliente, tam_cin) < 0){
                    perror ("Erro no envio de Mensagem apra cliente");
                    exit (1);
                }
            }
        }
    };
};

```

```

    fprintf ( stdout, "Estourou o timer, encaminha pacote de novo. \n");
    return 0;
};

void EnviaPacote (int socket, struct sockaddr_in cliente, char pacote[MAX_LINE+1]) {
    /* Funcao de envio do pacote de dados recuperados do arquivo solicitado */

    int tam_cin = sizeof ( cliente );
    fprintf ( stdout, "%d\n", strlen(pacote) );
    /* envia bloco de dados com numero da janela */
    if ( sendto (socket, pacote, strlen (pacote) +1, 0, (struct sockaddr *) & cliente, tam_cin) < 0 )
    {
        perror ("Erro no envio de pacote para cliente");
        exit (1);
    }

    fprintf (stdout, " Pacote %c enviado para cliente.\n ", pacote[0] );
};

int main (int argc, char * argv[]) {
    struct sockaddr_in sin; /* estrutura de endereço do servidor */
    struct sockaddr_in client; /* estrutura de endereço do cliente */
    char pacote[MAX_LINE+1]; /* pacote com o numero da janela (0 ou 1) e o buffer de no maximo 256
    bytes recuperado do arquivo desejado */
    char bloco[MAX_LINE]; /* buffer com string recuperada do arquivo solicitado pelo cliente */
    char NomeArq[30]; /* string com nome do arquivo solicitado */
    int eof = 0; /* variavel que indica final de arquivo */
    int ACK = 0; /* variavel que indica ACK recebido */
    int len;
    int tam_cin; /* tamanho da estrutura do cliente */
    int s;
    int W;
    int limite_tentativas;

    /* monta estrutura de dados do endereço */
    bzero ((char *) &sin, sizeof (sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons (SERVER_PORT);

    /* configura abertura passiva */
    if ((s= socket (AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror ("servidor arquivos: socket");
        exit (1);
    };

    if ((bind(s, (struct sockaddr*)&sin, sizeof(sin))) < 0 ) {
        perror("servidor arquivos: bind");
        exit (1);
    };

    /* entra em loop infinito para atender varios clientes, um por vez */
    while (1)
    {
        tam_cin = sizeof (client);

        fprintf ( stdout, "Aguardando novo cliente.\n");
        /* recebe primeiro pacote do cliente com nome do arquivo desejado */
        len = recvfrom(s, NomeArq, sizeof(NomeArq), 0, (struct sockaddr *) &client, &tam_cin);

        /* se len < 0 indica erro no recebimento */
        if (len < 0) {

```

```

        perror("servidor arquivo: recvfrom");
        exit (1);
    };
    fprintf (stdout, "Recebeu um pedido do arquivo %s.\n", NomeArq);

    /* Abre arquivo para leitura */
    eof = AbreArquivo ( s, client, NomeArq );

    W = 0; /* primeiro bloco a ser transmitido */
    while ( !eof ) { /* faca enquanto o arquivo nao acabar */
        /* recupera um bloco de pelo menos 256 bytes do Arquivo */
        eof = PegaBloco ( bloco );

        /* monta pacote que sera transmitido */
        MontaPacote ( &W, pacote, bloco );

        ACK = 0;
        limite_tentativas = 0;
        do {
            /* enquanto nao receber ACK retransmite o mesmo pacote varias vezes */
            EnviaPacote ( s, client, pacote );

            /* Aguarda ACK ou estoura o timeout */
            ACK = AguardaACK ( s, client );
            if ( ! ACK && limite_tentativas > 15 ) {
                fprintf ( stdout, "Excedeu 16 tentativas de envio do mesmo pacote.\n");
                ACK = 1;
                eof = 1;
            }
            limite_tentativas = limite_tentativas + 1;

        } while ( ! ACK ); /* faca enquanto nao receber confirmacao */
    } /* faca enquanto o arquivo nao terminar */
    if ( Arq != NULL ) fclose (Arq);
}; /* fim da comunicacao com um cliente */
};

```