UNIVERSIDADE DO ESTADO DO AMAZONAS - UEA

ESCOLA SUPERIOR DE TECNOLOGIA

ENGENHARIA DE COMPUTAÇÃO

CLARICE DE SOUZA SANTOS

IMPLEMENTAÇÃO DO PROTOCOLO TFTP EM ERLANG

Manaus

2011

CLARICE DE SOUZA SANTOS

IMPLEMENTAÇÃO DO PROTOCOLO TFTP EM ERLANG

Trabalho de Conclusão de Curso apresentado à banca avaliadora do Curso de Engenharia de Computação, da Escola Superior de Tecnologia, da Universidade do Estado do Amazonas, como pré-requisito para obtenção do título de Engenheiro de Computação.

Orientador: Prof. M.Sc. Jucimar Maia da Silva Júnior

Manaus

2011

Universidade do Estado do Amazonas - UEA
Escola Superior de Tecnologia - EST

Reitor:
José Aldemir de Oliveira
Vice-Reitor:
Marly Guimarães Fernandes Coelho
Diretor da Escola Superior de Tecnologia:
Mário Augusto Bessa de Figueirêdo
Coordenador do Curso de Engenharia de Computação:
Danielle Gordiano Valente
Coordenador da Disciplina Projeto Final:
Raimundo Correa de Oliveira

Banca Avaliadora composta por: *Data da Defesa: 16/12/2011.*
Prof. M.Sc. Jucimar Maia da Silva Júnior *(Orientador)*
Prof. M.Sc. Raimundo Corrêa de Oliveira
Prof. M.Sc. Ernande Ferreira de Melo

CIP - Catalogação na Publicação

S237i SANTOS, Clarice de Souza

Implementação do Protocolo TFTP em Erlang / Clarice de Souza Santos; [orientado por] Prof. MSc. Jucimar Maia da Silva Júnior - Manaus: UEA, 2011.

240 p.: il.; 30cm

Inclui Bibliografia

Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação). Universidade do Estado do Amazonas, 2011.

CDU: 004

CLARICE DE SOUZA SANTOS

IMPLEMENTAÇÃO DO PROTOCOLO TFTP EM ERLANG

Trabalho de Conclusão de Curso apresentado à banca avaliadora do Curso de Engenharia de Computação, da Escola Superior de Tecnologia, da Universidade do Estado do Amazonas, como pré-requisito para obtenção do título de Engenheiro de Computação.

Aprovado em: 16/12/2011

BANCA EXAMINADORA

Prof. Jucimar Maia da Silva Júnior, Mestre
UNIVERSIDADE DO ESTADO DO AMAZONAS

Prof. Raimundo Corrêa de Oliveira, Mestre
UNIVERSIDADE DO ESTADO DO AMAZONAS

Prof. Ernande Ferreira de Melo, Mestre
UNIVERSIDADE DO ESTADO DO AMAZONAS

Agradecimentos

Agradeço primeiramente a Deus que sempre me deu forças e me guiou nos momentos mais difíceis, não me deixando desistir jamais.
Agradeço também a minha mãe que sempre lutou e se sacrificou para que eu pudesse estudar. Mãe consegui!!!
Aos meus familiares e amigos, em especial, ao João Guilherme que me apoiou e ajudou sempre que necessário.
Aos meus professores da UEA que sempre me ajudaram. Principalmente ao professor Jucimar Maia que aceitou ser meu orientador, obrigada pela paciência, compreensão, direcionamento e por sempre acreditar em mim, mesmo nos momentos que eu mesma tinha dúvidas que conseguiria.
Por fim, e não menos importante, agradeço ao Rafael Augusto que compreendeu minha ausência e sempre esteve ao meu lado.

Resumo

O TFTP (Trivial File Transfer Protocol) é um protocolo de transferencia de arquivos que foi projetado para ser fácil de manusear. Ele funciona sobre o UDP e por isso implementa todo o processamento de detecção e recuperação de perdas de informação ao nível da aplicação. Este artigo propõe a implementaçãodo protocolo através do servidor e cliente TFTP.

Abstract

The TFTP (Trivial File Transfer Protocol) is a file transfer protocol that is designed to be easy to handle. He works over UDP, so implements the process of error detection and recovery of lost information at the application level. This paper proposes the implementation of the protocol through the TFTP server and client.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Códigos

Capítulo 1

Introdução

As linguagens de programação têm o propósito de produzir softwares, portanto, toda linguagem tem por objetivo principal o desenvolvimento de programas ou sistemas de computação, conforme as características de seu projeto. [da Silva Vieira2011]

A linguagem Erlang é uma linguagem funcional para programação de sistemas concorrentes e distribuídos, que foi desenvolvida pela Ericsson e Ellemtel Computer Science Laboratories. Foi projetada para programação concorrente, tempo real e sistemas distribuídos tolerante a falhas. [Armstrong et al.2007]

O protocolo TFTP(Trivial File Transfer Protocol) é um protocolo muito simples usado para transferir arquivos, a única coisa que pode fazer é ler e escrever arquivos de/para um servidor remoto.

Dentre os vários própositos do Erlang está a implementação de protocolos. O protocolo escolhido para ser implementado neste trabalho foi o TFTP, que é simples e pode demonstrar a capacidade do Erlang em implementar protocolos.

1.1 Objetivo

Implementar o protocolo TFTP em Erlang

1.1.1 Objetivos específicos

- Desenvolver um cliente TFTP;
- Desenvolver um servidor TFTP;

1.2 Metodologia

O desenvolvimeto do protocolo inicia com o estudo do protocolo TFTP, para o melhor entendimento do seu funcionamento, e a modelagem do sistema.

A implemetação começará com o estabelecimento da arquitetura do cliente e servidor que somente estabelece a conexão entre eles. Posteriormente será implemetado o envio e recebimento dos pacotes de escrita, leitura, dados e confirmação.

Depois a utilizaçao de temporizadores será colocada, junto com a retransmissão dos pacotes de dados e confirmação. Por fim o envio e processamento de pacotes de erro é feito.

Depois de implementado, o sistema será testado com servidor e cliente existente. A metodologia utilizada foi:

- Estudo do Protocolo
- Modelagem do sistema
- Arquitetura cliente e servidor - conexão
- Envio e processamento dos pacotes WRQ e RRQ
- Envio e processamento dos pacotes DAT e ACK
- Ativação e expiração de temporizadores
- Retransmissão de pacotes DAT e ACK
- Envio e processamento de pacotes de erro
- Teste com servidor TFTP existente
- Teste com cliente TFTP existente

1.3 Organização do trabalho

No capítulo 2 são apresentados os protocolos IP (*Internet Protocol*), UDP (*User Datagram Protocol*) e TFTP (*Trivial File Transfer Protocol*).

No capítulo 3 é definido a linguagem de programaçao Erlang.

No capítulo 4 é apresentado o desenvolvimento.

No fim do trabalho é feita uma conclusão sobre a implentação feita.

Capítulo 2

Protocolos

Um protocolo é um padrão utilizado para controlar conexão, comunicação ou transferência de dados entre dois sistemas. Ou seja, um protocolo pode ser definido como um conjunto de regras que define como a comunicação acontece no que diz respeito a sintaxe, semântica e sincronização da comunicação.

2.1 Protocolo IP (Internet Protocol)

O Internet Protocol (IP) é um protocolo de comunicação descrito no RFC 791 da IETF em Setembro de 1981. O IP é projetado para comutação de pacotes em redes de computadores, transmite blocos de dados chamados datagramas da origem até o destino.

O protocolo IP não garante entrega confiável, os pacotes podem chegar desordenados, duplicados, ou podem ser perdidos por inteiro. Não existem mecanismos para aumentar a a confiabilidade dos dados, o controle de fluxo, sequenciamento, mas ele pode usar os serviços fornecidos pela própria rede de apoio para garantir qualidades de serviço.

Um módulo de internet está presente em cada host que participa da comunicação e também em cada gateway que interliga as redes, esses módulos possuem regras para interpretar campos de endereço, para fragmentar e montar datagramas e também regras para tomar decisões de roteamento (a seleção de um caminho para a transmissão é chamada de roteamento) .

Os ”módulos de internet”utilizam os endereços contidos no cabeçalho de internet para transmitir os datagramas para o destino, e usam os campos no cabeçalho para fragmentar e remontar os datagramas quando necessário para a transmissão através de ”pequenos pacotes”de redes.

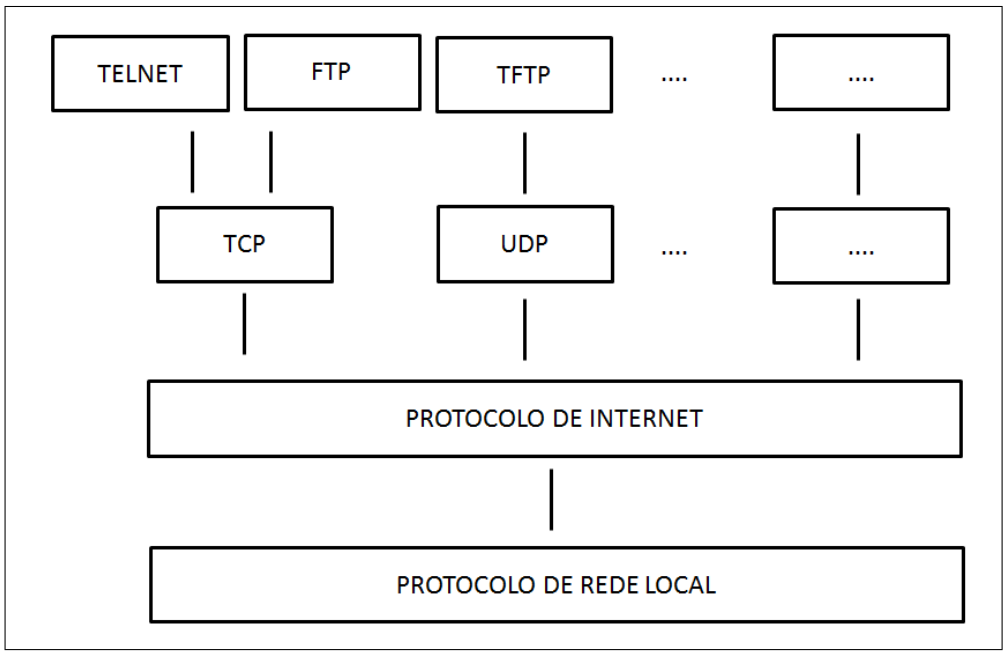

Figura 2.1: Hierarquia de protocolos

O IP não fornece comunicação confiável, não há confirmações de recebimento, controle de erro para os dados, apenas o checksum no cabeçalho, não há retransmissões nem controle de fluxo.

2.1.1 Modelo de operação

O IP se comunica de um lado com protocolos de alto nível host-to-host e do outro com protocolos da rede local. O IP é chamado pelo protocolo de alto nível e solicita aos protocolos de rede local para enviar o datagrama para o próximo gateway ou o host de destino. Por exemplo, o TCP chama o IP passando o endereço e outros parâmetros, o IP cria o datagrama e chama a interface de rede local para transmitir o datagrama. A hierarquia dos protocolos pode ser vista na figura 2.1.

O caminho de transmissão de um datagrama é exemplificado na figura 2.2. Por exemplo, para enviar um datagrama de uma aplicaçao para outra os seguintes passos podem ser seguidos:

- O programa de origem prepara os dados e chama o módulo de internet para enviar os dados como um datagrama, passando o endereço de destino e outros parâmetros como argumentos da chamada;
- O módulo de internet prepara o cabeçalho do datagrama e anexa os dados nele;

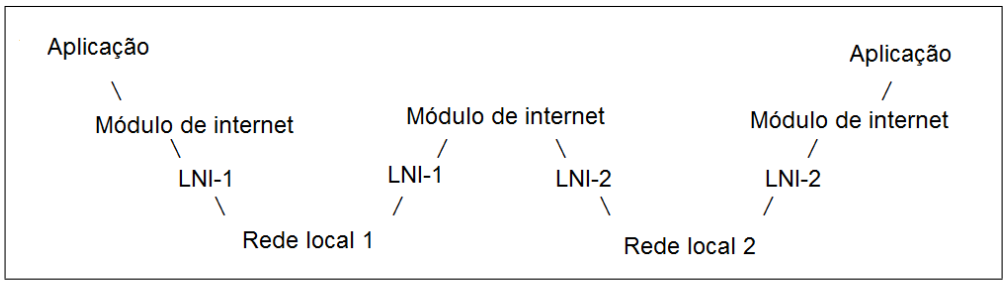

Figura 2.2: Caminho de transmissão do datagrama

- O módulo de internet determina um endereço de rede local para este endereço de internet (neste caso, é o endereço de um gateway) e envia o datagrama e o endereço de rede local para a interface de rede local;
- A interface de rede local cria um cabeçalho de rede local e anexa o datagrama nele, então envia o resultado pela rede local;
- O datagrama chega ao gateway contido no cabeçalho da rede local, retira o cabeçalho da rede local e encaminha o datagrama ao seu módulo de internet;
- O módulo de internet identifica apartir do endereço de internet que o datagrama deve ser encaminhado para outro host em uma segunda rede. O módulo de internet então gera um endereço de rede local para o host de destino e chama a interface de rede local para enviar o datagrama;
- A interface de rede local cria um cabeçalho de rede local, anexa o datagrama e envia o resultado para o destino;
- No destino o cabeçalho de rede local é retirado pela interface de rede local e entreque para módulo de internet;
- O módulo de internet identifica que o datagrama é para um programa no próprio host e passa os dados para a aplicação em resposta a uma chamada do sistema, passando o endereço de origem e outros parâmetros como resultado da chamada.

2.1.2 Função

O IP implementa duas operações básicas:

- Endereçamento - A função do IP é mover datagramas através de redes. O envio é feito passando os datagramas de um módulo de internet para outro até o destino,

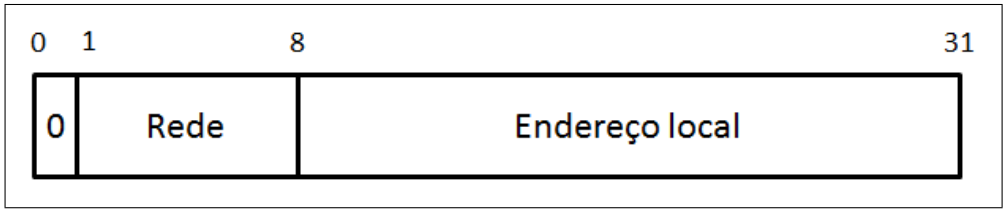

Figura 2.3: Classe A

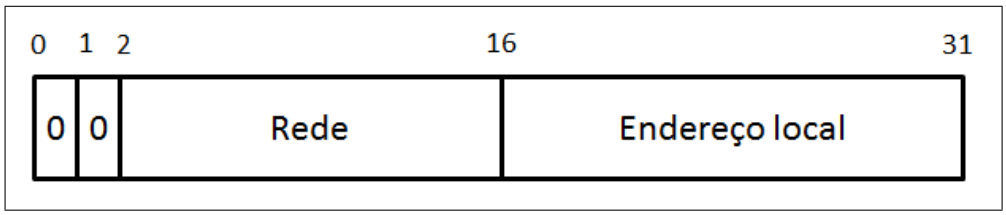

Figura 2.4: Classe B

apartit da interpretação do endereço de internet. Assim, um importante mecanismo do protocolo de internet é o endereço de internet.

- Fragmentação - No envio de mensagens de um módulo de internet para outro, os datagramas podem precisar atravessar uma rede cujo tamanho máximo do pacote é menor que o tamanho do datagrama. Para superar esta dificuldade, um mecanismo de fragmentação é fornecido no protocolo de internet.

Endereçamento

É feita uma distinção entre os nomes, endereços e rotas. Um nome indica o que buscamos, um endereço indica onde está e uma rota indica como chegar lá. O IP trabalha com endereços, os protocolos de alto nível mapeiam nomes para endereços, o móduulo de internet mapeia endereços de internet para endereços de rede local e os protocolos de baixo nível mapeiam endereços de rede local para rotas.

Os endereços tem o tamanho de quatro octetos (32 bits). Um endereço começa com um número de rede, seguido pelo endereço local.Existem 3 classes para o endereço de internet:r

- Classe A - o bit de alta ordem é zero, os próximos 7 bits são a rede, e os últimos 24 bits são o endereço local. A classe A é mostrada na figura 2.3;

- Classe B - os dois bits de alta ordem dois zero, os próximos 14 bits são a rede e os últimos 16 bits são o endereço local. A classe B é mostrada na figura 2.4;

- Classe C - os três bits de alta ordem são 110, os próximos 21 bits são a rede e os últimos 8 bits são os endereço local. A classe C é mostrada na figura 2.5.

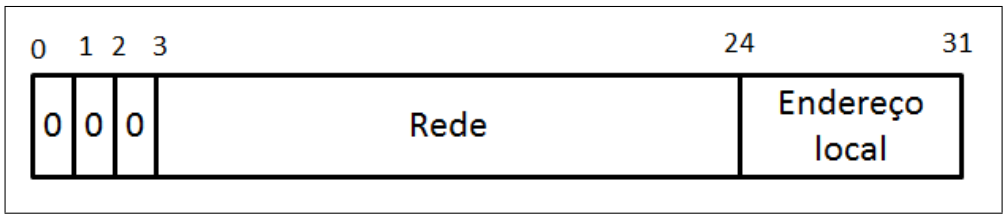

Figura 2.5: Classe C

Fragmentação

A fragmentaçao é necessária quando um datagrama passa por uma rede local na qual o limite do tamanho de um pacote é menor que o limite da sua rede de origem. Um datagrama pode ser marcado para não ser fragmentado, se for marcado e a rede necessitar fragmentar, mas não puder o datagrama é descartado.

O processo de fragmentação deve ser capaz de quebrar o datagrama pedaços de possam ser remontados. O receptor do datagrama usa o campo de idenntificação para que pedaços de diferentes datagramas não sejam unidos. Os campos offset e tamanho de cada fragmento indicam a posição do fragmento no datagrama original.

Para fragmentar um longo datagrama o módulo de internet cria dois novos datagramas e copia o conteúdo do cabeçalho do datagrama original para os cabeçalhos dos dois novos. Os dados são divididos em duas partes, a primeira é formada por 8 octetos (64 bits), a segunda pode não ser multiplo de 8 octetos, mas a primeira deve ser. A primeira parte dos dados é colocada no primeiro datagrama, o campo tamanho é preenchido com o tamanho do primeiro datagrama. e a flag de mais fragmentos é colocado para 1. A segunda parte dos dados é colocada no segundo datagrama, o campo tamanho é preenchido com o tamanho do segundo datagrama e a flag de mais datagramas é preenchida com o mesmo valor do datagrama original. O campo offset é prenchido com o valor desse campo do datagrama original mais o número do fragmento. Esse método pode ser generalizado para n partiçoes do datagrama original.

Para montar os fragmentos um módulo de protocolo de internet junta os datagramas que têm o mesmo valor para os quatro campos: identificação, origem, destino e protocolo. A montagem é feita unindo os dados de cada fragmento na ordem indicado pelo campo offset do cabeçalho, o primeiro fragmento tem o offset igual a zero e o último tem a flag de mais fragmentos igual a 0.

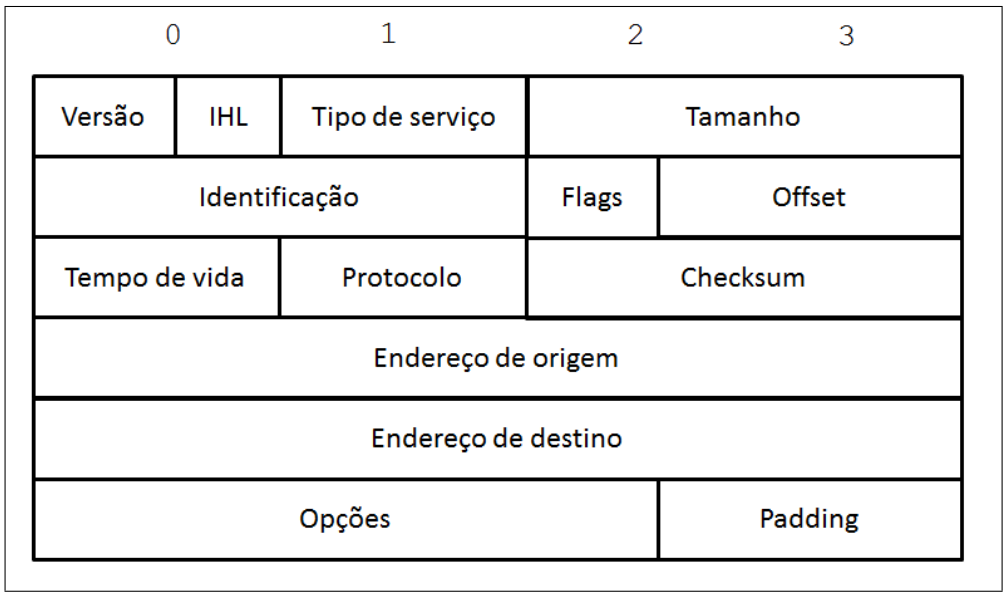

Figura 2.6: Cabeçalho IP

2.1.3 Formato do cabeçalho IP

O cabeçalho do protocolo IP mostrado na figura 2.6 é formado pelos campos:

- Versão (4bits)- indica a versão do formato do cabeçalho;
- Tamanho (4 bits) - é o tamanho do cabeçalho em palavras de 32 bits, aponta para o início dos dados;
- Tipo de serviço (8 bits)- é um conjunto de parâmetros que determinam as escolhas de serviço prestado nas redes que compõem a internet, é usado para indicar a qualidade desejada para o serviço. Fornece uma indicação dos parâmetros da qualidade de serviço desejada na transmissão de um datagrama através de uma rede particular.A escolha deve levar em consideração o baixo atraso, alta confiabilidade e alto rendimento. É usado por gateways para selecionar os parâmetros de transmissão da rede, a próxima rede a ser utilizada, ou o próximo gateway de roteamento. O formato do campo é mostrado na figura 2.7, os bits do campo são:
  - Bit 0-2 - Precedência. 111=Controle de Rede, 110=Controle Internetwork, 101=CRÍTICO/ECP, 100=Flash Override, 011=Flash, 010=Imediata, 001=Prioridade, 000=Rotina;
  - Bit 3 - Delay.0=normal, 1=baixo;
  - Bit 4 - Throughput. 0=normal, 1=alto;
  - Bit 5 - Confiabilidade. 0=normal, 1=alta;

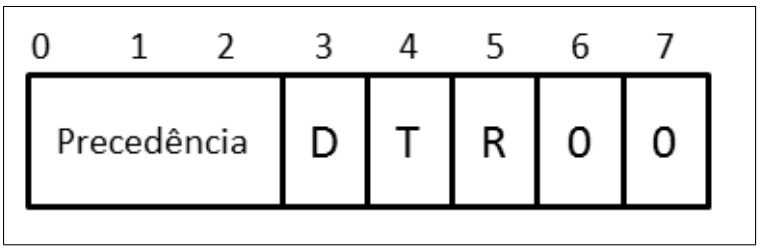

Figura 2.7: Campo Tipo de serviço

  - Bit 6-7 - Reservado para uso futuro.

- Tamanho (16 bits) - é o tamanho do datagrama, medido em octetos, inclue cabeçalho de internet e dados. O tamanho máximo de um datagrama é 65.535 octetos.
- Identificação - um valor que identifica unicamente o datagrama para ajudar na remontagem de fragmentos.
- Flags - Os bits do campo são:
  - Bit 0 - reservado, deve ser 0;
  - Bit 1 - 0=Fragmentado, 1=Não fragmentado;
  - Bit 2 - 0= Útimo fragmento, 1= Tem mais fragmentos;
- Offset - indica qual a posição medido em 8 octetos do fragmento no datagrama. O primeiro fragmento tem offset igual a 0;
- Tempo de vida (Time to Live) - é o tempo limite de vida de um datagrama. É definido pelo remetente e reduzido em cada ponto da rede onde passa, se o tempo chega a zero antes do datagrama chegar no destino ele é destruído.
- Protocolo - permite saber de que protocolo procede o datagrama ;
- Checksum - fornece uma verificação de que os dados foram transmitidos corretamente. Se o cheksum do cabeçalho falhar o datagrama é descartado pela entidade que detectou o erro.
- Endereço de origem - endereço IP da origem;
- Endereço de destino - endereço IP do destino;
- Opcões - incluem disposições para timestamps, segurança e roteamento especial.

- Padding - é usada para garantir que o cabeçalho da internet termina em um limite de 32 bits. O preenchimento é zero.

2.2 Protocolo UDP (User Datagram Protocol)

O User Datagram Protocol (UDP) é um protocolo da camada de transporte usado com o protocolo IP da camada de rede. O UDP é definido pela RFC 768 escrita por John Postel.

O serviço prestado pelo UDP não é confiável, pois não garante a entrega dos pacotes e não trata duplicação. O UDP fornece um mínimo, não confiável, melhor esforçoo, de transmissão de mensagens de transporte para aplicações e protocolos de camada superior. [Fairhurst].

A comunição pelo UDP não necessita de estabelecimento de conexão, oferece um eficiente transporte de comunicação para algumas aplicações, mas não provê controle de congestionamento ou confiabilidade.

O UDP é definido para disponibilizar um datagrama para troca de pacotes entre computadores em uma rede. O protocolo disponibiliza um método para que as aplicações possam enviar mensagens para outros programas.

O UDP é orientado a transação, por isso não garante a entrega dos pacotes(o protocolo que garante a entrega é o TCP). O UDP não garante segurança nas comunicações, as aplicações que necessitem proteger sua comunicação devem utilizar mecanismos de protocolos adicionais.

2.2.1 Formato do cabeçalho

Um computador pode enviar datagramas sem estabelecer conexão com o destinatário. Um datagrama UDP é feito em um único pacote IP e, portanto, é limitado a uma carga máxima de 65.507 bytes para IPv4 e IPv6 para 65.527 bytes. [Fairhurst]. A transmissão de grandes pacotes necessitam de fragmentação, o que reduz a confiabilidade e eficiência da comunicação.

Para enviar um datagrama, é preciso preencher os campos do cabeçalho UDP e encaminhar os dados junto com o cabeçalho do protocolo IP da camada de rede.

O cabeçalho UDP mostrado na figura 2.8 é composto por quatro campos de 2 bytes cada, sendo eles: porta de origem, porta de destino, tamanho e cheksum.

- Porta de origem: é opcional, quando utilizada representa a porta remetente que pode

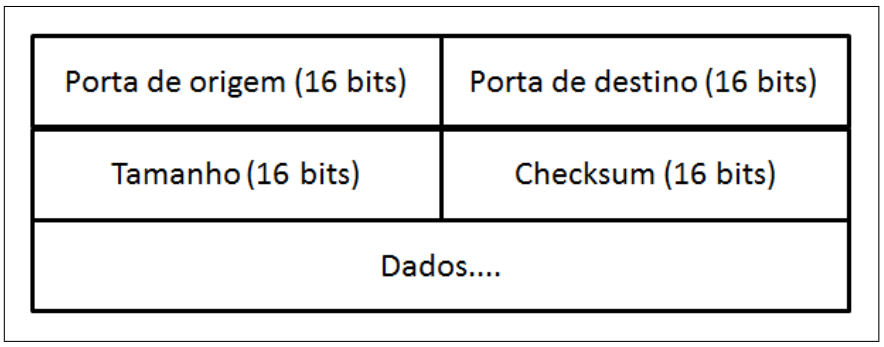

Figura 2.8: Formato do cabeçalho UDP

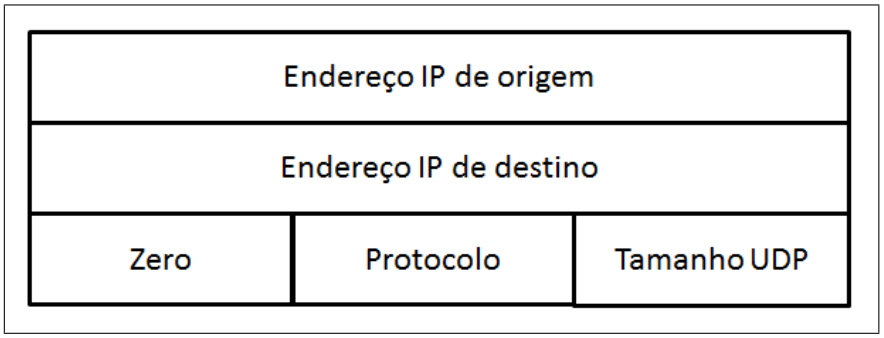

Figura 2.9: Pseudo cabeçalho prefixado ao UDP

ser usada para receber uma resposta, se não for usada seu valor padrão é 0(zero);

- Porta de destino: é a porta de destino, indica o serviço solicitado;
- Tamanho: é o tamanho do datagrama, inclui o cabeçalho e os dados. (seu valor mínimo é oito);
- cheksum: é uma verificação do pacote que é feito para garantir que os dados não foram corrompidos por roteadores ou pontes nas redes. Serve também para o receptor verificar se ele é mesmo o destino do pacote. Se o transmissor não gerar o cheksum seu valor será 0 (zero).

Quando chega no destino a camada do protocolo UDP recebe o pacote da camada de rede, verifica o pacote com o cheksum (se ele for maior que 0) e descarta todos os pacotes inválidos. UDP não provê nenhuma forma de reportar erros se o pacote não for entregue.

O cabeçalho que é colocado antes do cabeçalho do UDP contém o endereço de origem, endereço de destino, o protocolo utilizado e o tamanho. O pseudo cabeçalho prefixado ao UDP é mostrado na figura 2.9.

2.2.2 Interface de usuário

A interface deve permitir:

- Criação de novas portas de recebimento;
- Operações de recebimento nas portas que retornem dados e uma indicação do endereço e porta de origem;
- Uma operação que permita enviar um datagrama, especificando os dados, portas de origem e destino e os endereços para envio.

2.2.3 Interface IP

O módulo UDP deve ser capaz de determinar o endereço de origem e destino e os campos do protocolo do cabeçalho de internet. Uma interface UDP/IP retorna o datagrama de internet completo incluindo todos os cabeçalhos em resposta a uma operação de recebimento. Essa interface também permite o UDP passar um datagrama de internet completo com cabeçalho para o IP enviar. O IP verifica a consistência de alguns campos e calcula o cheksum.

2.2.4 Aplicação do protocolo

Como o UDP não fornece nehuma confiabilidade, como não retransmitir perda de pacotes, não tratar recebimento de datagramas duplicado, não garantir que os pacotes cheguem em ordem. Se a aplicação que o usa exigir entrega confiável, o próprio programa deve implementar mecanismos de verificação e retransmissão de pacotes, como por exemplo o TFTP faz. Os maiores usos do UDP são na Internet Name Server e no Trivial File Transfer Protocol (TFTP).

2.3 TFTP (Trivial File Transfer Protocol)

O protocolo TFTP(Trivial File Transfer Protocol) é um protocolo muito simples usado para transferir arquivos, definido pela RFC 1350 de Julho de 1992. A única coisa que pode fazer é ler e escrever arquivos de/para um servidor remoto, não pode listar diretórios ou fazer autenticação do usuário.

O TFTP permite a transferência de arquivos entre um cliente TFTP e um servidor TFTP [TechNet] .O protocolo TFTP é implementado usando o UDP para enviar e receber dados e implementa seu próprio esquema de confiabilidade usando UDP.

Três modos de transferência são suportados *netcascci* para arquivos de texto, *octet* para binários e *mail* que já é considerado obsoleto.

2.3.1 Funcionamento

Servidores TFTP escutam na porta 69 e após receber uma requisição de um cliente, o servidor aloca randomicamente uma porta para a transmissão entre eles. A troca inicial de pacotes permite que o servidor mova o cliente para uma porta diferente.

Qualquer transferência de arquivos utilizando o protocolo é iniciada por um pedido de leitura(RRD) ou escrita(WRQ) do cliente para o servidor, que também funciona como pedido de conexão. Se o servidor "aceita"o pedido, a conexão é aberta e o arquivo é transferido em blocos de tamanho fixo de 512 bytes.

Cada pacote de dados contém um bloco de dados e deve ser reconhecido por um pacote de acknowledgment antes que o próximo pacote possa ser enviado. Se um pacote se perde na rede, o destinatário espera um tempo limite e retransmite seu último pacote (que pode ser dados ou um acknowledgment), fazendo que o remetente do pacote perdido retransmita o pacote perdido. Um pacote de dados menor que 512 bytes sinaliza o término de uma transferência [Gavidia].

O remetente precisa guardar apenas um pacote para retransmissão, pois a confirmação de recebimento dos pacotes anteriores garante que todos os pacotes mais antigos tenham sido recebidos. Ambas as máquinas envolvidas na transferência são consideradas remetentes e receptores, um envia dados e recebe confirmação, a outra envia confirmação e recebe dados.

O TFTP é muito restritivo, a fim de simplificar a implementação. Por exemplo, a confirmação fornece controle de fluxo e elimina a necessidade de reordenar os pacotes de dados.

Quando um cliente deseja ler um arquivo ele envia um RRQ para o porta 69 do computador remoto e quando quer escrever envia WRQ. O pacote inclui *opcode*, o nome do arquivo terminado com um byte 0 e o modo que pode ser *netcascci* para arquivos de texto, *octet* para binários e *mail* que já é considerado obsoleto.

Na leitura o servidor responde com um pacote de dados com número de bloco 1 e para escrita é retornado um ACK. O servidor responde ao cliente por uma porta diferente da

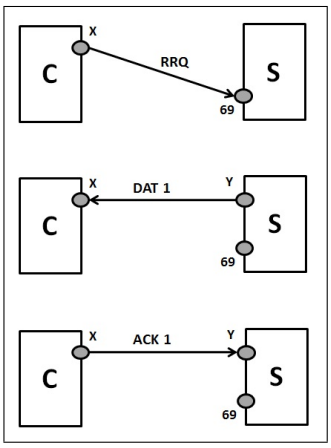

Figura 2.10: Funcionamento da leitura

69 e o cliente passa a usar essa nova porta para futuras transmissões.

O emissor deve esperar por um tempo pelo ACK, senão receber ele transmite o pacote novamente. Caso o receptor já tenha recebido o pacote antes, o ACK se perdeu, ele deve descartar o pacote e transmitir o ACK novamente [Williams].

Leitura

O funcionamento da solicitação de leitura é mostrado na figura 2.10 e segue os passos:

- Cliente envia um RRQ para o porta 69;
- Servidor responde com um pacote de dados com número de bloco 1;
- O servidor responde ao cliente por uma porta diferente da 69 e o cliente passa a usar essa nova porta para futuras transmissões.

Escrita

O funcionamento da solicitação de escrita é mostrado na figura 2.11 e segue os passos:

- Cliente envia um WRQ para o porta 69;
- É retornado um ACK;
- O servidor responde ao cliente por uma porta diferente da 69 e o cliente passa a usar essa nova porta para futuras transmissões.

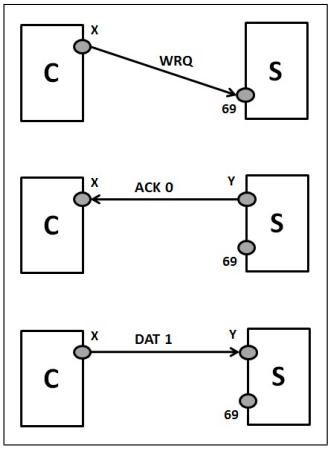

Figura 2.11: Funcionamento da escrita

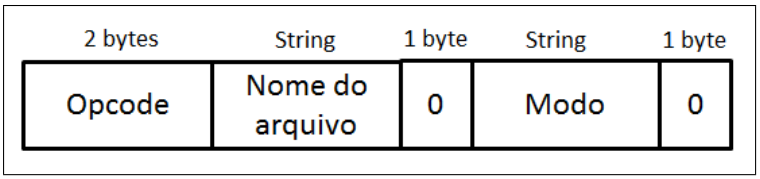

Figura 2.12: Pacote RRQ e WRQ

2.3.2 Pacote TFTP

Cada datagrama tem um opcode (2bytes) que informa o tipo de pacote está sendo transmitido. Existem 5 tipos de pacotes [Williams], como mostrado na tabela 2.1 :

Código	Nome	Descrição
1	RRQ	Solicita leitura de arquivo
2	WRQ	Solicita leitura de arquivo
3	DAT	Dados do arquivo
4	ACK	Confirmação
5	ERR	Erro

Tabela 2.1: *Tipos*

Os pacotes do protocolo TFTP podem ser [Gavidia]:

- Pacote RRQ, WRQ : Formado pelo código do pacote, nome do arquivo que é uma sequência de bytes em netascii terminando por um byte zero e o modo terminado por 0. O formato do pacote é mostrado na figura 2.12.
- Pacote DATA : Formado pelo código do pacote, número do bloco e os dados. O formato do pacote é mostrado na figura 2.13.
- Pacote ACK: Formado pelo código do pacote e número do bloco. O formato do pacote é mostrado na figura 2.14.

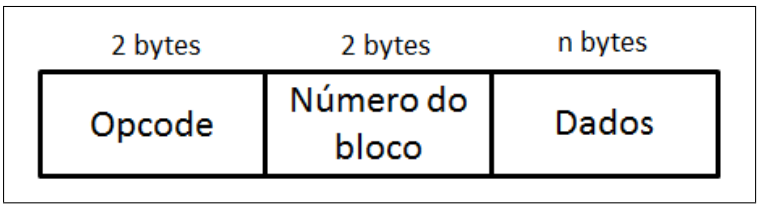

Figura 2.13: Pacote Dados

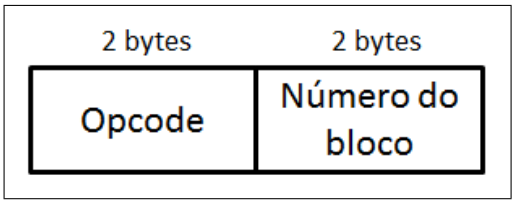

Figura 2.14: Pacote ACK - confirmação

- Pacote Error:Formado pelo código do pacote,código do erro e mensagem de erro terminada com byte 0. O formato do pacote é mostrado na figura 2.15.

2.3.3 Erros

A maioria dos erros causam o término da conexão. Um erro é sinalizado enviando um pacote de erro, este pacote não é confirmado e não é retransmitido. Portanto timeouts são usados para detectar o término quando o pacote de erro foi perdido.

Erros são causados por três tipos de eventos: não ser capaz de satisfazer o pedido (por exemplo, arquivo não encontrado, violação de acesso), recebendo um pacote que não pode ser explicado por um atraso ou duplicação da rede (por exemplo, um pacote formado incorretamente), e perder o acesso a um recurso necessário (por exemplo, disco cheio ou acesso negado durante uma transferência).

Os tipos de erros possíveis são mostrados na tabela 2.2 [Williams]:

O TFTP reconhece somente uma condição de erro que não causa término de conexão a porta de origem de um pacote recebido está incorreto. Neste caso, um pacote de erro é enviado para o host de origem.

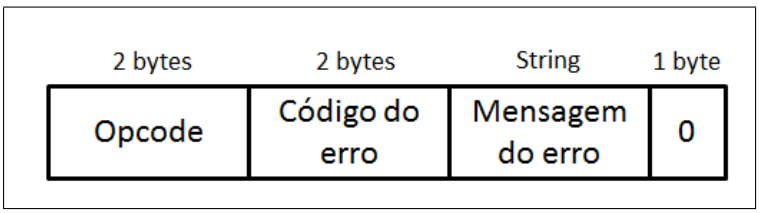

Figura 2.15: Pacote de erro

Código	Descrição
0	Indefinido
1	Arquivo não encontrado
2	Violação de acesso
3	Disco cheio
4	Operação ilegal
5	ID de transferência desconhecido
6	Arquivo já existe
7	Arquivo não existe

Tabela 2.2: *Erros*

Capítulo 3

Erlang

Em meados dos anos 80 a Ericssom fez várias experiências com algumas linguagens de programação para identificar os requisitos necessários para o desenvolvimento de softwares para telecomunicação. Como nenhuma linguagem existente reunia todas as características necessárias, a Ericsson resolveu desenvolver uma linguagem que atendesse essas características e a necessidade de desenvolver sistemas complexos de telecomunicações.

Erlang é uma linguagem funcional para programação de sistemas concorrentes e distribuídos, que foi desenvolvida pela Ericsson e Ellemtel Computer Science Laboratories. Foi projetada para programação concorrente, tempo real e sistemas distribuídos tolerante a falhas. [Armstrong et al.2007]

O Erlang possibilita programação de sistemas concorrentes de tempo real em alto nível, permite a redução no esforço requerido para projetar, implementar e manter aplicações. Tem como características:

- Sintaxe declarativa;
- Concorrência - tem um modelo baseado em concorrência com troca de mensagem assíncrona.
- Tempo real - destinado a programação de sistemas onde o tempo de resposta exigido é de milisegundos.
- Operação continua - tem primitivas que permitem a substituição de código enquanto o sistema está rodando e permite também que antigas e novas versões do código executem ao mesmo tempo. Essa característica é interessante para sistemas que não

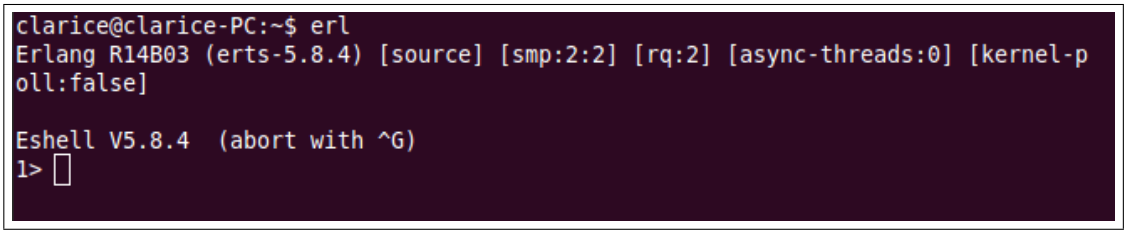

Figura 3.1: Shell do Erlang

podem parar de rodar para que alterações sejam feitas, como por exemplo sistemas de tráfego aéreo.

- robusto - existem três metódos para detectar erros em tempo de execução que podem ser usados em aplicações robustas que não permitem falhas.

- Gerenciamento de memória - possui um garbage collector (coletor de lixo) que aloca memória quando necessário e desaloca quando não é mais utilizada.

- Distribuição - Erlang não compartinha memória, toda a comunicação entre processos é feita apartir de trocas de mensagens assíncronas. Uma aplicação feita para um processador pode ser portada para uma rede de processadores, essa característica faz o Erlang construir facilmente sistemas distribuídos.

- Integração - possibilita usar programas escritos em outras linguagens, como se fossem escritos em erlang.

3.1 Shell

Erlang tem seu próprio shell onde é possível escrever e executar diretamente código Erlang e ver seu resultado. O shell interpreta comandos inseridos pelo usuário, permite declaração de variáveis e chamada de funções de programas Erlang, só não é possível definir uma função no shell. O Shell solicita uma expressão, executa e imprime o resultado [Armstrong et al.2007].

Para iniciar o shell é preciso abrir o interpretador de comandos do sistema operacional e digitar *erl*. A figura 3.1 mostra um exemplo de inicialização de shell.

O Erlang shell numera as linhas nas quais podem ser inseridos comandos (1> 2>). Para finalizar a entrada de uma linha de código é necessário acrescentar um ponto final ”.”indicando que a expressão acabou. Se algo for digitado errado é possível deletar caracteres com a tecla bakspace.

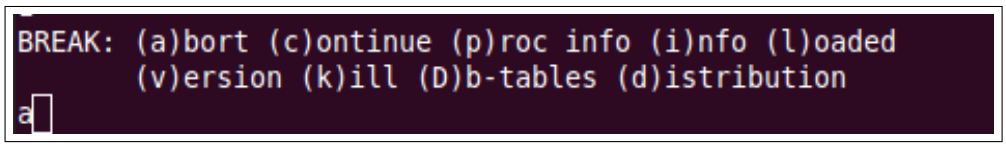

Figura 3.2: Finalizar o shell

Para finalizar o shell basta digitar Control+C,a figura 3.2 mostra a saída que aparecerá, então digita- se "a"e deixa o sistema Erlang, outra maneira de finalizar é digitando *halt().*

3.2 Programação sequencial

Todas as funções Erlang pertencem a algum módulo particular. O módulo mais simples possível contém módulo de declaração, declarações de exportação e código representando as funções que são exportados do módulo. Funções exportadas podem ser executadas de fora da módulo, todas as outras funções somente executam dentro do módulo. [Armstrong et al.2007]

Um exemplo de código em Erlang utilizando módulos e funções pode ser visto no código 3.2.1.

```
1  -module(fat).
2  -export([fatorial/1]).
3
4  fatorial(0) -> 1;
5
6  fatorial(N) -> N * fatorial(N-1).
```

Código 3.2.1: *Utilizando módulos e funções em Erlang*

No exemplo a função fatorial tem duas cláusulas, a primeira cláusula fatorial(0) que é terminada com ";"indicando que a função ainda tem mais partes e a segunda cláusula terminada com "."que indica o fim da funçao. Quando a função fatorial é executada com um argumento, as duas cláusulas são verificadas sequencialmente na ordem em que aparecem no módulo até que uma seja compatível com a chamada. Quando a compatibilidade ocorre a expressao do lado direito do símbolo ->"é executada. [Armstrong et al.2007]

Os programas Erlang são escritos em arquivos, cada arquivo contém um módulo, o nome do arquivo é o nome do módulo com a extensão ".erl". A primeira linha do módulo é uma declaração de módulo *-module(fat).* que define o nome do módulo. A segunda linha *-export([fatorial/1]).*, contém o nome e o número de argumentos das funções que são exportadas, ou seja, estão no módulo e podem ser chamadas de fora do módulo. As duas linhas, *module* e *export* iniciam com o caracter -"que é chamado de prefixo de atributo,

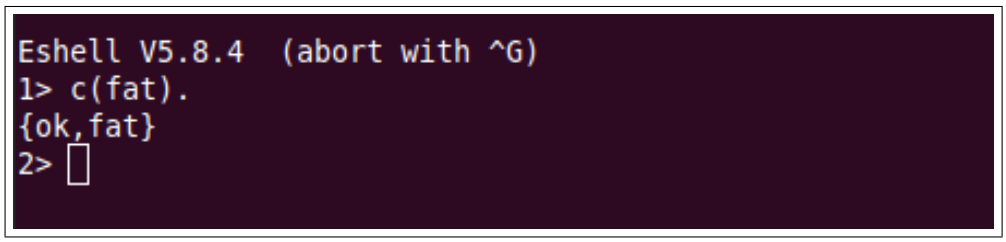

Figura 3.3: Shell do Erlang - Retorno da compilação

devem ser terminadas com ponto ".".Para usar a função em outro módulo basta usar a sintaxe "nome_do_módulo:nome_da_funçao(argumentos)". [Ericsson]

Para compilar o código em Erlang basta abrir o shell no mesmo diretório do arquivo e digitar *c(nome_do_programa)*. Se a compilação terminar com sucesso a mensagem {*OK,nome_do_modulo*} aparecerá, caso contrário uma mensagem de erro é retornada. A figura 3.3 mostra um exemplo de compilação.

Um exemplo de programa com duas funções exportadas, a primeira com 1 argumento e a segunda com 2 é mostrado no código 3.2.2.

```
1 -module(operacoes).
2 -export([dobro/1, soma/2]).
3
4 dobro(N) -> 2*N.
5
6 soma(X,Y) -> X+Y.
```

Código 3.2.2: *Exportação de funções*

3.3 Variáveis

As variáveis são usadas para armazenar valores de tipos de dados, o nome de uma variável é definido com a primeira letra maiúscula seguida de letras maiúsculas, minúsculas ou números, não pode conter caracteres especial exceto o underscore. Não é necessário declarar uma variável explicitamente (definir seu tipo), o tipo das variáveis e a viabilidade das operacões são determinados durante a execução.

Não existe variável global em Erlang, todas as variáveis são consideradas locais na função. As variáveis em Erlang possuem atribuição única, ou seja, quando se atribui um valor para uma variável esse valor não pode ser alterado, caso alguma operação seja feita com o valor da variável o resultado deve ser atribuido a outra variável. A figura 3.4 mostra um erro de atribuição única.

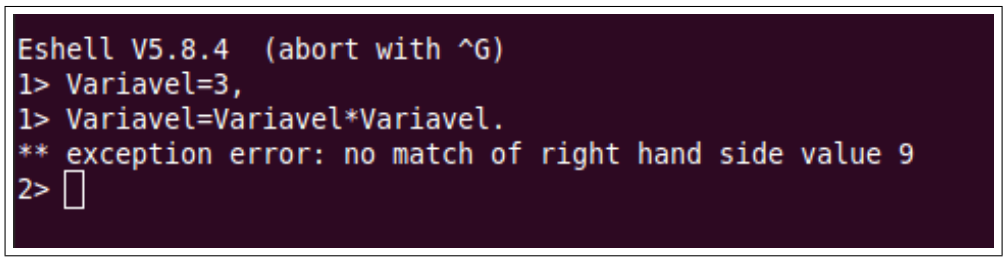

Figura 3.4: Erro de atribuição única

3.4 Tipos de dados

Os tipos de dados suportados pelo Erlang chamados de Terms são:

- Constantes
  - Números
    - Integer - guarda números naturais
    - Float - para guardar números reais
  - Atom - são apenas constantes com nomes
  - Pids (identificador de processo) - para guardar nomes de processos
  - Reference - para guardar referência única do sistema
- Compostos
  - Tuples, para guardar um número fixo de termos
  - Lists, para guardar um número variável de termos

3.4.1 Interger

Integers em Erlang são usados para representar números inteiros positivos ou negativos e podem ser expressos em bases diferentes de 10. A precisão de inteiros é uma questão local, mas a precisão de pelo menos 24 bits deve ser fornecida por qualquer sistema Erlang.

A notação $<char> representa o valor ASCII do caractere Char assim, por exemplo, $A representa os 65 inteiro. Inteiros com base diferente de 10 são escritos usando a notação <Base>#<Valor> assim, por exemplo, 16#ffff representa os 65.535 inteiro (na base 10). O valor da Base de Dados deve ser um inteiro na faixa de 2 .. 16. [Armstrong et al.2007]

3.4.2 Float

Floats em Erlang são usados para representar números reais com ponto flutuante. A notação NE-P é a notação convencional de declaração de ponto flutuante, onde N é o número real e P é o ponto flutuante. Em 1234E-10 por exemplo, o ponto é movido 10 casas decimais a esquerda, sendo o mesmo que escrever $1234\text{x}10^{-10} ou 0 : 0000000001234.1001$[*da Silva Vieira*2011]

3.4.3 Atom

Atoms são constantes com nomes, o "valor"de um atom é o nome que lhe é atribuído e as únicas operaçõoes possíveis com atoms são comparações. São simples nomes, não são como variáveis que podem ser atribuido um valor.

Atoms iniciam com letra minúscula como por exemplo: segunda, centimetro. Se for colocado entre apóstrofe qualquer caracter pode ser usado em um átomo. Se for usado apóstrofe as regras mostradas na tabela 3.1 são seguidas [Armstrong et al.2007]:

Caracter	**Significado**
\b	backspace
\d	deletar
\e	escape
\f	form feed
\n	nova linha
\r	retorno
\t	tab
\ \	contra barra
\ A.. \ Z	control A para control Z
\’	apóstrofe
\”	aspas
\000	caracter com representação octal

Tabela 3.1: *Convenção de Atoms*

3.4.4 Pid

Os programas Erlang executados não compartilham parâmetros ou memória, eles são independentes, sendo assim cada thread é um processo. Como os processos não compartilhan memória a única ligação entre os processos é o identificador de processos (PID).

Em Erlang os PIDs são usados para a comunicação entre processos através da troca de mensagens e para identificá-los no ambiente de desenvolvimento, permitindo operações como a criação, destruição e troca de mensagens entre processos em execução [da Silva Vieira2011].

3.4.5 Reference

Reference é um term que é único no sistema Erlang criado chamando *make_ref*, serve como um atalho. A funçao *make_ref* retorna um objeto único garantindo que será diferente de todos os outros objetos do sistema.

3.4.6 Tuple

Tuples são terms separados por vírgula e delimitados por chave, usadas para armazenar um número fixo de itens que podem ser de tipos diferentes, são similares a *records* e *structures*.

Os terms indivivuais que estão nas tuples são chamados de elementos. O tamanho de uma tuple é o número de elementos que ela possui, por exemplo a tuple {E1,E2,...,En} tem tamanho igual a n. Os elementos são identificados pela posição que estão na tuple.

Quando em uma tuple o primeiro elemento é um atom, este atom é chamado de tag. Esta convençao do Erlang é usada para distinguir entre tuples com diferentes propósitos dentro de algum código [da Silva Vieira2011].

Exemplos e tuples:

{a , 12 , 'oi'}
{1 , 2 , {3 , 4} , {a , {b , c}}}
{}

3.4.7 List

Lists são terms separados por vírgula e delimitados por colchete, usadas para armazenar um número variável de itens que podem ser de tipos diferentes. O tamanho de uma list é o número de elementos que ela possui, por exemplo a list [E1,E2,...,En] tem tamanho igual a n.

Exemplos de lists:

[1, abc, [12], 'foo bar']
[a,b,c]
"abcd"

A notação entre aspas (") que é chamada de string é uma abreviação para a representação ASCII da lista de caracteres dentro das aspas. Por exemplo "abc"significa [97,98,99].

Por convenção o primeiro elemento da lista é chamada de cabeça e o resto da lista é chamado de cauda. A list [E1, E2, E3, ...,En|Variavel] onde n>=1 significa que os primeiros

n elementos da lista são E1, E2, E3, ..., E4 e o resto da lista é a variável Variavel.

3.5 Sistema de módulo

Erlang tem um sistema de módulo que permite que um grande programa seja dividido em pequenos módulos. Cada módulo tem seu próprio espaço de nome, o que permite que o mesmo nome de funçao seja usado em diferentes módulos.

O sistema de módulo trabalha limitando a visibidade das funções que estão no módulo. A maneira pela qual uma função pode ser chamada depende do nome do módulo, o nome da função e se o nome da função ocorre em uma declaração de importação ou exportação no módulo. [Armstrong et al.2007]

```
1  -module(lists1).
2  -export([reverse/1]).
3
4  reverse(L) ->
5          reverse(L, []).
6
7  reverse([H|T], L) ->
8          reverse(T, [H|L]);
9
10 reverse([], L) ->
11         L.
```

Código 3.5.1: *Módulo em Erlang*

O programa 3.5.1 define uma função reverse/1 que reverte a ordem dos elementos da lista. A única função que pode ser acessada de fora do módulo é a *reverse/1*. As funções que podem ser chamadas de fora do módulo devem estar na declaração *export* do módulo. As outras funções *reverse/2* estão disponível apenas dentro do módulo.

As funções *reverse/1* e *reverse/2* são funções diferentes, pois em Erlang duas funções com o mesmo nome, mas com número de argumentos diferente são funções diferentes.

3.5.1 Chamadas internas dos módulos

Existem duas formas de chamar funções em um módulo. Ambas as formas são usadas para evitar ambiguidade, quando por exemplo dois módulos exportão funçoes com mesmo nome. As duas formas são:

- Usando o caminho completo da funçao nome_do_modulo:nome_da_funcao(parametros), como é mostrado no código 3.5.2.
- Usando um nome implicito através da declaração *import*, depois de importar uma função de um módulo externo é possível chama-lá somente pelo nome, como mostrado no código 3.5.3.

```
1  -module(sort1).
2  -export([reverse_sort/1, sort/1]).
3
4  reverse_sort(L) ->
5          lists1:reverse(sort(L)).
6
7  sort(L) ->
8          lists:sort(L).
```

Código 3.5.2: *Chamada de função externa nome_do_modulo:nome_da_funcao(parametros)*

```
1  -module(sort2).
2  -import(lists1, [reverse/1]).
3  -export([reverse_sort/1, sort/1]).
4
5  reverse_sort(L) ->
6          reverse(sort(L)).
7
8  sort(L) ->
9          lists:sort(L).
```

Código 3.5.3: *Chamada de função externa através da declaração import*

3.5.2 Cláusulas

Cada funçao em Erlang possue um número de cláusulas que são separadas por ponto e vírgula ";". Cada cláusula é composta de head (cabeça), uma opcional guard (guarda) e um body (corpo).

Head

O head é o nome da função seguido pelo número de argumentos separados por vírgula.

Guard

Guard são condições que devem ser satisfeitas antes de uma cláusula ser escolida, pode ser um simples teste ou uma sequencia de testes separados por vírgula.

Para avaliar uma guard, todos os seus testes são avaliados. Se todos são verdadeiros, então o guard tem sucesso, caso contrário é uma falha. A ordem de avaliaçao dos testes em uma guard é indefinida. Se a guard tem sucesso, então o corpo da cláusula é avaliado. Se o teste falha, a próxima cláusula é testada. [Armstrong et al.2007]

Quando o head e o guard são satisfeitos o body da cláusula selecionada é executado. Um exemplo so uso de guardvpode ser visto no código 3.5.4.

```
1  factorial(N) when N == 0 -> 1;
2
3  factorial(N) when N > 0 -> N * factorial(N - 1).
```

Código 3.5.4: *Uso de Guard*

Os possíveis testes de guarda são mostrados na tabela 3.2:

Guard	Sucesso se
atom(X)	X é um atom
constant(X)	X não é uma list ou tuple
float(X)	X é um float
integer(X)	X é um integer
list(X)	X é uma list ou []
number(X)	X é um integer ou float
pid(X)	X é um identificador de processo
port(X)	X é uma porta
reference(X)	X é uma referência
tuple(X)	X é uma tupla
binary(X)	X é um binário

Tabela 3.2: *Testes de guard*

As operações de comparação permitidas em um guard são mostradas na tabela 3.3:

Operador	Descriçao
X > Y	X maior que Y
X < Y	X menor que Y
X =< Y	X menor ou igual Y
X >= Y	X maior ou igual Y
X == Y	X igual Y
X /= Y	X diferente de Y
X =:= Y	X exatamente igual Y
X =/= Y	X exatamente diferente de Y

Tabela 3.3: *Operações de guard*

Body

O corpo de uma cláusula consiste em uma sequência de um ou mais expressões que são separadas por vírgula ",". As expressões de uma cláusula são executadas na sequência em que aparecem, o valor da cláusula é o valor da última expressao.

Existem várias razões para dividir o corpo de uma função em uma sequência de chamadas: [Armstrong et al.2007]

- Para garantir a execuçao sequencial do código - cada expressao no body da função é executada sequencialmente.
- Aumentar a clareza - é mais fácil escrever uma funçao como uma sequência de expressões.
- Para desempacotar valores de retorno da função.
- Reutilizar o resultado de uma chamada de função, código 3.5.5.

```
1  good(X) ->
2          Temp = lic(X),
3          {cos(Temp), sin(Temp)}.
```

Código 3.5.5: *Reutilização de resultado de função*

3.6 Programação Concorrente

Processos e comunicação entre processos são conceitos funcamentais em Erlang e toda concorrência, criação de processos e comunição entre eles são explícitas. [Armstrong et al.2007]

3.6.1 Criação de processos

Um processo é uma unidade auto-suficiente, separada da computação que existe concorrentemente com outros processos no sistema. A função *spawn/3* cria e inicia a execução de novos processos, a sintaxe é a seguinte:

Pid=spawn(nome_do_modulo, nome_da_função, lista_de_argumentos)

Ao invés de executar a função e retornar o resultado, o *spawn/3* cria um proceso concorrente para executar a função e retorna o Pid do novo processo que é usado para todas as formas de comunicação dos processos. O *spawn/3* não espera a excecução da função, retorna logo após criar o processo. O processo termina automaticamente assim que a execução da função acaba, mas o retorno da função é perdido.

3.6.2 Comunicação entre processos

A Única forma de comunicação entre processos em Erlang é a troca de mensagens. Uma mensagem é enviada através da primitiva "!"(enviar): Pid ! Mensagem

Pid é o identificador do processo para o qual a mensagem é enviada, a mensagem pode ser qualquer term válido de Erlang e "!"é uma primitiva que avalia seus argumentos. Seu valor de retorno é a mensagem enviada. [Armstrong et al.2007]

O envio de mensagens é uma operação assíncrona, o remetente não espera a mensagem chegar no destino. Se o processo destino já tiver acabado nenhuma mensagem de erro é retornada para o remetente. As mensagens são sempre entregues na ordem que foram enviadas.

Para receber uma mensagem é usada a primitica *receive*, a sintaxe é mostrada no código 3.6.1.

Cada processo tem uma caixa postal na qual são armazenadas as mensagens enviadas ao processo na mesma ordem de chegada. As mensagens da caixa postal são "comparadas"com

```
1  receive
2          Mensagem1 [when guard1]-> Ação1;
3          Mensagem2 [when guard2]-> Ação2;
4          .....
5  end
```

Código 3.6.1: *Primitiva receive*

os padrões contidos no *receive*, quandoocorre o casamento de uma mensagem, e a execução da guard correspondente tem sucesso, a mensagem é selecionada, removida da caixa postal e então a ação correspondente é executada. As mensagens não "atendidas"pelo receive permanecem na caixa postal, e não são retiradas. [Armstrong et al.2007]

Receber mensagens de um processo específico

Para receber mensagens de um processo epecífico, o processo remetente deve enviar o seu identificador na mensagem. Para enviar o identificador o processo pode usar a função self() que retorna o identificador do processo, a mensagem enviada é *Pid ! {self(),Mensagem}* . O destino deve esperar a mensagem identificando o Pid do processo, como no código 3.6.2.

```
1  receive
2          {Pid, Msg}->
3          .....
4  end
```

Código 3.6.2: *Receber mensagens de um processo específico*

3.6.3 Registro de processo

Para enviar uma mensagens para um processo é preciso saber o identificador do processo, mas em grandes sistemas esse método não é prático. Para permitir o envio de mensagens sem saber o Pid do processo é possível registrar o processo dando-lhe um nome, apartir daí é permitido enviar mensagens usando o nome do processo. As funções que fazem a manipulação do registro de processos são:

- register(Nome, Pid) - associa um nome atom com um Pid.
- unregister(Nome) - remove a associação entre atom e Pid.
- whereis(Nome) - retorna o Pid associado ao atom, se nehum processo está associado o retorno é o atom *undefined*.
- registeres() - retorna uma lista com todos o nome de todos os atoms registrados.

O envio de mensagem pode ser feito utilizando o nome registrado *nome_atom ! Mensagem*

3.7 Programação com sockets

Um socket é um ponto de comunicação que permite a comunicação entre máquinas através da Internet usando o Internet Protocol (IP) [Armstrong2007]. Erlang tem duas bibliotecas para programação com sockets: *gen_tcp* para programar com aplicações TCP e *gen_udp* para programação UDP.

Sockets em Erlang podem ser abertos em três modos: *active*, *active once* ou *passive*, a escolha do modo de abertura é feita incluindo a opção {*active, true* | *false* | *once*} no argumento *Opcoes* em *gen_tcp:connect(Endereco, Porta, Opcoes)* e em *gen_tcp:listen(Porta, Options)*. Os modos são especificados com as opções:

- {*active, true*} - cria um socket ativo;
- {*active, false*} - cria um socket passivo;
- {*active, once*} - cria um socket que é ativo, mas somente para o recebimento de uma mensagem, após receber a mensagem o socket deve ser ativado novamente antes de receber a próxima mensagem.

A diferença entre um socket ativo e o passivo acontece quando uma mensagem é recebida. O socket ativo não cosegue controlar o fluxo de dados recebidos, um cliente pode enviar milhares de mensagens para o sistema e o processo de controle não consegue parar o fluxo de mensagem, podendo ocorrer perda de informação por sobrecarga. O socket passivo consegue controlar o fluxo de dados por que a leitura das mensagens não é feita automaticamente, uma mensagem só é lida com a chamada a função *gen_tcp:recv(Socket, N)*, a função tenta receber N bytes do socket assim o servidor pode controlar o fluxo de mensagens do cliente, escolhendo quando chamar *gen_tcp:recv*.

3.7.1 Socket usando TCP

Para abrir um socket TCP é usada a função *connect/3* do módulo *gent_tcp*. Os argumentos da funçao são: endereço , porta de destino e opções *connet(endereco, porta, opcoes)*. O retorno da função é {*ok,Socket*} se o socket foi aberto com sucesso, sendo "Socket"o socket aberto, ou {*error,Razao*} caso um erro tenha ocorrido, sendo "Razao"uma mensagem de erro. Um exemplo é mostrado no código 3.7.1.

No código 3.7.1 um socket é aberto na porta 80 com a função *gen_tcp:connet*. A opção *binary* abre o socket no modo binário e entrega todos os dados da aplicação como binário, a opção {*packet,0*} significa que os dados são entregues para a aplicaçao sem nenhuma

```
1  nano_get_url(Host) ->
2          {ok,Socket} = gen_tcp:connect("www.google.com",80,[binary, {packet, 0}]),
3          ok = gen_tcp:send(Socket, "GET / HTTP/1.0\r\n\r\n"),
4          receive_data(Socket, []).
5
6  receive_data(Socket, SoFar) ->
7          receive
8                  {tcp,Socket,Bin} ->
9                          receive_data(Socket, [Bin|SoFar]);
10                 {tcp_closed,Socket} ->
11                         list_to_binary(reverse(SoFar))
12         end.
13
14
```

Código 3.7.1: *Socket TCP*

modificação. Para enviar uma mensagem pelo socket é usada a funçao *send/2* que tem como argumentos o socket e a mensagem a ser enviada *send(Socket, Mensagem)*. O retorno da funçao *send* é o atom *ok* caso o envio tenha ocorrido com sucesso ou {*error,Razao*} caso tenha ocorrido um erro. O retorno recebido pelo socket pode ser de dois tipos:

- {*tcp, Socket, Bin*} - o primeiro argumento é o atom *tcp*, o segundo é o socket *Socket* e o terceiro é um binário, no caso do socket ser aberto no modo binário.
- {*tcp_closed, Socket*} - o primeiro argumento da list é o atom *tcp_closed*, o segundo é o socket. Esse retorno significa que o socket foi fechado.

A aplicação servidor fica escutando em uma porta, esperando as conexões. A função que espera a conexão é *listen/2* que tem como argumentos a porta e as opções *listen(Porta, Opcoes)*. O retorno da função é {*ok, Socket*}, onde *Socket* é o socket lido ou {*error, Razao*}. Um exemplo de servidor é mostrado no código 3.7.2.

```
1  start_nano_server() ->
2          {ok, Listen} = gen_tcp:listen(2345, [binary, {packet, 4},{reuseaddr, true},{active, true}]),
3          {ok, Socket} = gen_tcp:accept(Listen),
4          gen_tcp:close(Listen),
5          loop de leitura......
```

Código 3.7.2: *Servidor usando socket TCP*

A conexão com o socket lido no *listen* é feito com a funçao *accept* que aceita a comunicaçao com o socket. A *accept/1* tem como argumento o socket lido em *listen*.

3.7.2 Socket usando UDP

UDP é um protocolo que não necessita estabecer conexão, o que significa que o cliente não tem que estabelecer uma conexão com o servidor antes de enviar mensagens. [Armstrong2007]

Para abrir um socket UDP é usada a função *open* do módulo *gen_udp*. A função *open/1,2* poder ser usada com um ou dois argumentos, sendo o primeiro a porta e o segundo uma lista de opções. O retorno da função é {*ok,Socket*} se o socket foi aberto com sucesso, sendo "Socket"o socket aberto, ou {*error,Razao*} caso um erro tenha ocorrido, sendo "Razao"uma mensagem de erro. Um exemplo é mostrado no código 3.7.3.

```
1  server(Port) ->
2          {ok, Socket} = gen_udp:open(Port, [binary]),
3          loop(Socket).
4
5  loop(Socket) ->
6          receive
7                  {udp, Socket, Host, Port, Bin} ->
8                          gen_udp:send(Socket, Host, Port, Mensagem),
9                          loop(Socket)
10         end.
```

Código 3.7.3: *Socket UDP*

No código 3.7.3 o socket é aberto no modo binário, o que significa que todos os dados são enviados como binário. Para enviar uma mensagem é usada a função *send/4* que tem como argumentos o socket, o endereço, a porta e a mensagem a ser enviada *send(Socket, Endereco, Porta, Mensagem)*. O retorno da funçao *send* é o atom *ok* caso o envio tenha ocorrido com sucesso ou {*error,Razao*} caso tenha ocorrido um erro.

Os dados recebidos pelo socket UDP tem o formato {*udp, Socket, Host, Port, Bin*}, o primeiro argumento é um atom *udp*, o segundo é o socket, o terceiro é a máquina de origem, o quarto é a porta e o quinto são os dados em binário.

Capítulo 4

Desenvolvimento do protocolo TFTP

Este capítulo descreve a implementação do protocolo TFTP em Erlang, mostra os diagramas e codificação da aplicação desenvolvida.

4.1 Casos de uso

O diagrama de caso de uso descreve as funcionalidades propostas para um sistema. Os casos de uso do sistema são:

- Programa cliente - funcionalidades que um usuário pode solicitar do software cliente (leitura e escrita de um arquivo), mostrado na figura 4.1.
- Programa servidor - funcionalidades que o software cliente pode solicitar do software servidor (leitura RRQ e escrita WRQ de um arquivo), mostrado na figura 4.2.

4.2 Diagrama de estados

Os diagramas de estados descrevem os estados e transições que o "sistema"passa durante a execução de uma tarefa. Os diagramas do sistema são: Cliente RRQ, Cliente WRQ,

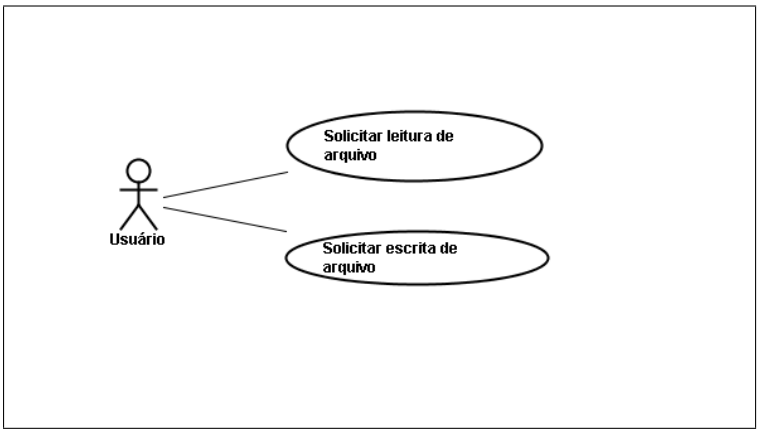

Figura 4.1: Caso de uso cliente

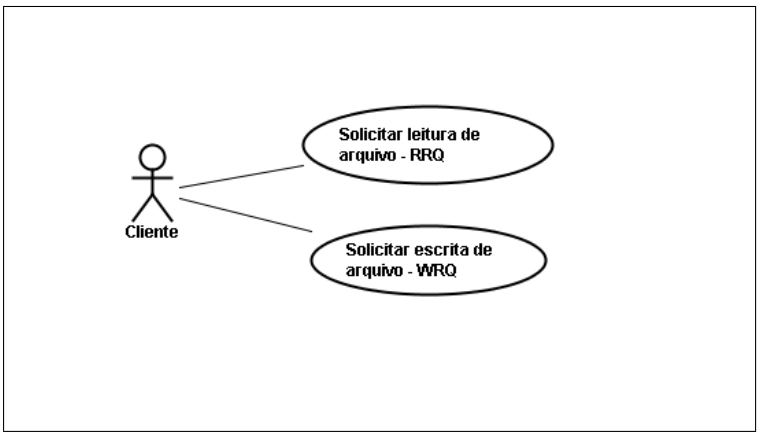

Figura 4.2: Caso de uso servidor

Servidor RRQ e Servidor WRQ.

4.2.1 Cliente RRQ

O diagrama de estados mostrado na figura 4.3 descreve os estados do cliente TFTP durante uma solicitação de leitura de um arquivo.

O cliente começa no estado inicial *Parado* até receber uma solicitação de leitura (RRQ) de um usuário entrando então no estado *Enviando RRQ*, quando termina de enviar a solicitação de leitura para o servidor entra no estado *Esperando resposta* no qual permanece esperando uma resposta do servidor para sua solicitação. A resposta do servidor pode ser um pacote de dados ou um mensagem de erro.

Quando recebe um pacote de dados do servidor, o cliente passa para o estado *Enviando Pacote* no qual verifica os dados recebidos pelo servidor e envia uma resposta que pode ser um pacote de erro ou uma confirmação de recebimento do pacote (ack) passando para o estado *Esperando resposta*, se o pacote de dados recebido for de tamanho menor que 512 bytes o cliente envia a confirmação e passa para o estado *Finalizando transação*, se algum erro ocorreu com o pacote de dados recebido uma mensagem de erro é enviada para o servidor e o cliente passa para o estado *Finalizando transação*

Se o servidor responde com uma mensagem de erro o cliente entra no estado *Finalizando transação* e termina a comunicação.

4.2.2 Cliente WRQ

O diagrama de estados mostrado na figura 4.4 descreve os estados do cliente TFTP durante uma solicitação de escrita de um arquivo.

O cliente começa no estado inicial *Parado* até receber uma solicitação de escrita (WRQ) de um usuário entrando então no estado *Enviando WRQ*. Quando termina de enviar a

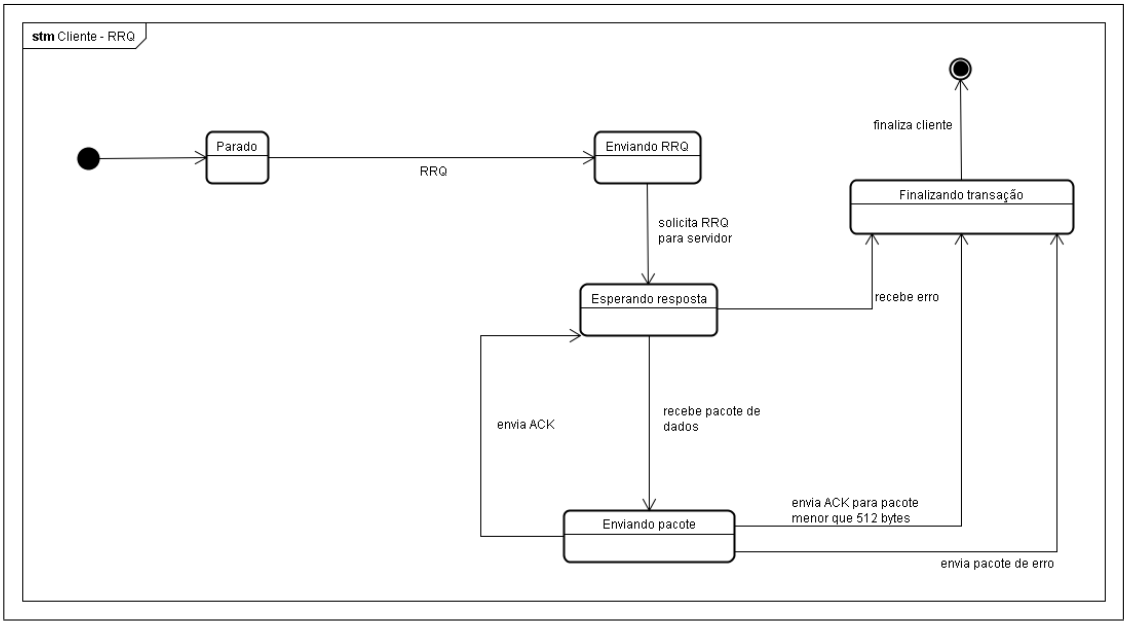

Figura 4.3: Diagrama de Estado - Cliente RRQ

solicitação de escrita para o servidor, o cliente entra no estado *Esperando resposta* no qual permanece esperando uma resposta do servidor para sua solicitação. A resposta do servidor pode ser um mensagem de erro ou uma confirmação com o número de pacote igual a zero, sinalizando a aceitação do pedido de escrita.

Quando recebe um pacote de confirmação do servidor, seja o inicial ou uma confirmação de uma pacote enviado anteriormente, o cliente passa para o estado *Enviando Pacote* no qual empacota os próximos dados do arquivo (512 bytes) e envia o pacote de dados para o servidor. Quando termina de enviar o pacote de dados o cliente passa para o estado *Esperando Resposta* no qual espera uma resposta do servidor para o pacote enviado.

A resposta pode ser um pacote de erro ou uma confirmação de recebimento do pacote (ack). Se a confirmação enviada pelo servidor for para um pacote de 512 bytes o cliente passa para o estado *Enviando Pacote*, mas se a confirmação for de um pacote de dados de tamanho menor que 512 bytes, significa que o último pacote de dados enviado foi o fim do arquivo e o cliente passa para o estado *Finalizando transação*. Se o servidor responde com uma mensagem de erro, o cliente entra no estado *Finalizando transação* e termina a comunicação.

Se algum erro ocorreu com o pacote de confirmaçao recebido pelo cliente, uma mensagem de erro é enviada para o servidor e o cliente passa para o estado *Finalizando transação*.

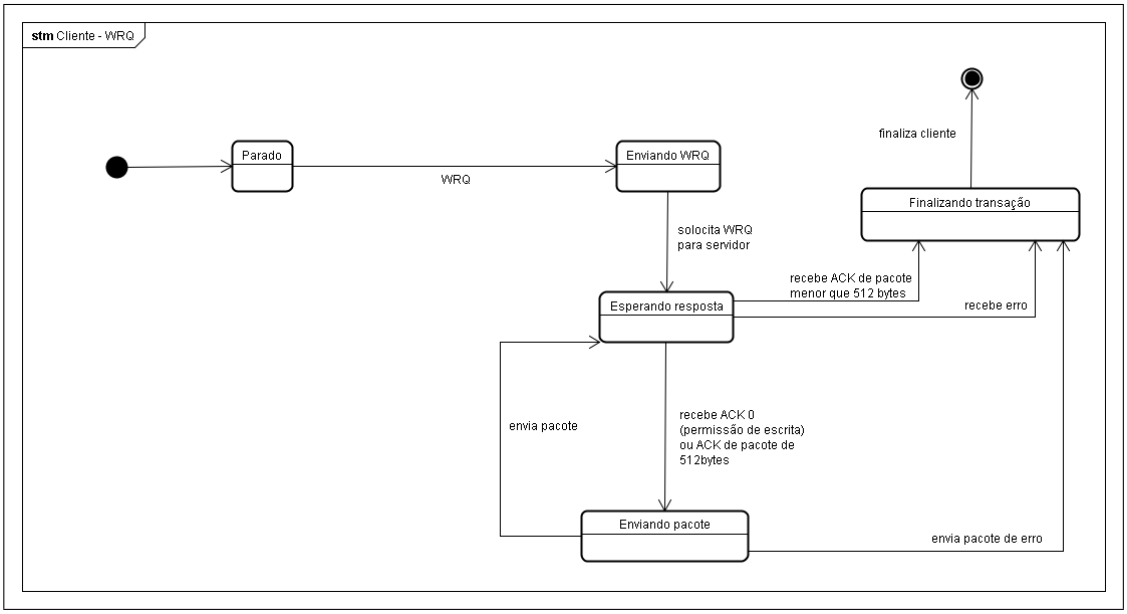

Figura 4.4: Diagrama de Estado - Cliente WRQ

4.2.3 Servidor RRQ

O diagrama de estados mostrado na figura 4.5 descreve os estados do servidor TFTP durante uma solicitação de leitura (RRQ) de um arquivo.

O servidor começa no estado inicial *Esperando solicitação na porta 69* até receber uma solicitação de leitura (RRQ) de um cliente e entra então no estado *Escolhendo nova porta para comunicação* no qual escolhe randomicamente uma nova porta, diferente da 69, para comunicação com o cliente.

Após escolher uma porta, o servidor entra no estado *Verificando existência do arquivo* onde verifica se o arquivo solicitado existe. Se não existir, o servidor envia uma mensagem de erro para o cliente e passa para o estado *Finalizando transação*. Se o arquivo existir, o servidor passa para *Enviando pacote*, onde os primeiros dados são empacotados e enviados para o cliente, após o envio o servidor vai para *Esperando resposta*.

A resposta do cliente pode ser um pacote de erro ou uma confirmação de recebimento do pacote (ack). Se a confirmação enviada pelo cliente for para um pacote de 512 bytes o servidor passa para o estado *Enviando Pacote* onde os próximos dados são empacotados e enviados para o cliente, mas se a confirmação for de um pacote de dados de tamanho menor que 512 bytes o servidor passa para o estado *Finalizando transação*. Se a resposta recebida pelo servidor for uma mensagem de erro, ele entra no estado *Finalizando transação* e termina a comunicação.

Se algum erro ocorreu com o pacote de confirmaçao, uma mensagem de erro é enviada para o cliente e o servidor passa para o estado *Finalizando transação*.

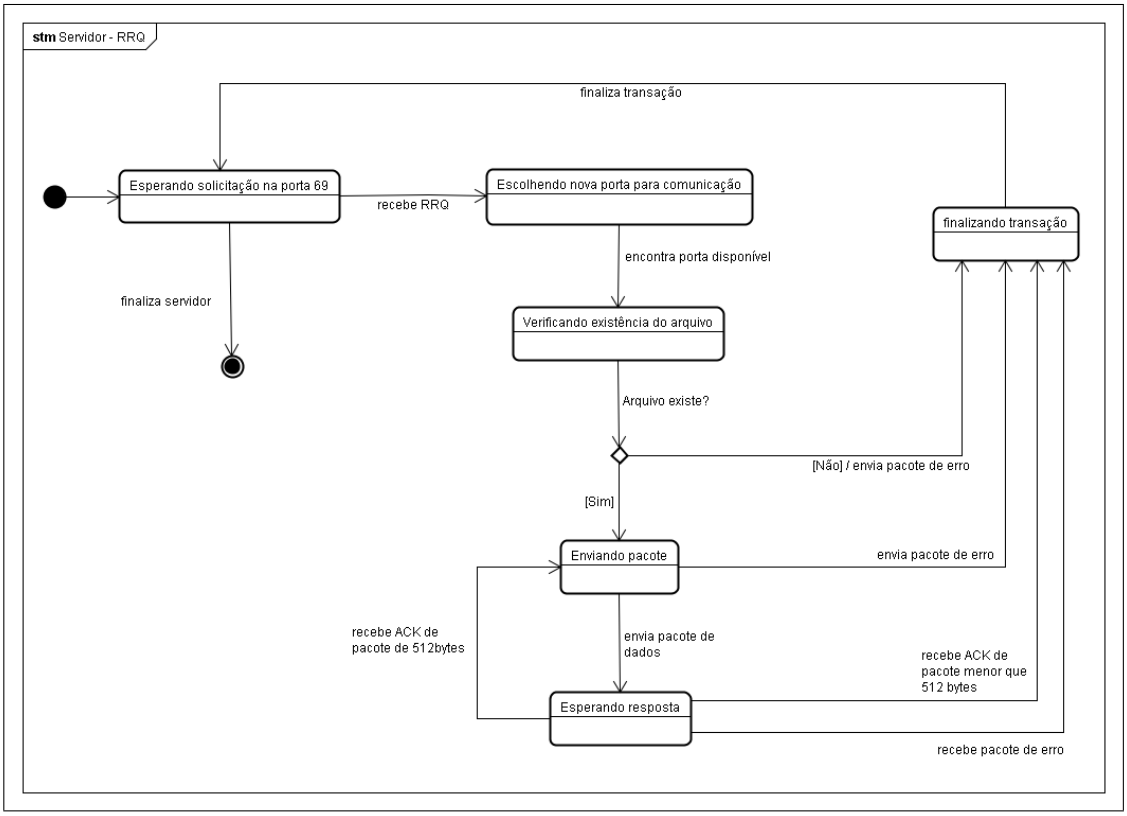

Figura 4.5: Diagrama de Estado - Servidor RRQ

4.2.4 Servidor WRQ

O diagrama de estados mostrado na figura 4.6 descreve os estados do servidor TFTP durante uma solicitação de escrita (WRQ) de um arquivo.

O servidor começa no estado inicial *Esperando solicitação na porta 69* até receber uma solicitação de escrita (WRQ) de um cliente e entra então no estado *Escolhendo nova porta para comunicação* no qual escolhe randomicamente uma nova porta, diferente da 69, para comunicação com o cliente.

Após escolher uma porta, o servidor vai para o estado *Verificando possibilidade de escrita* e verifica a possibilidade de escrita do arquivo. Se não não for possível, o servidor envia uma mensagem de erro para o cliente e passa para o estado *Finalizando transação*. Se o arquivo poder ser escrito, o servidor passa para *Enviando pacote* onde envia um pacote de confirmaçao com número de bloco igual a 0 para o cliente permitindo a escrita, após o envio o servidor vai para *Esperando resposta*.

A resposta do cliente pode ser um pacote de erro ou um pacote de dados. Quando recebe um pacote de dados do cliente, o servidor passa para o estado *Enviando Pacote* no qual verifica os dados recebidos pelo cliente e envia uma resposta que pode ser uma confirmação de recebimento do pacote ou um pacote de erro. Se o pacote de dados recebido for de 512

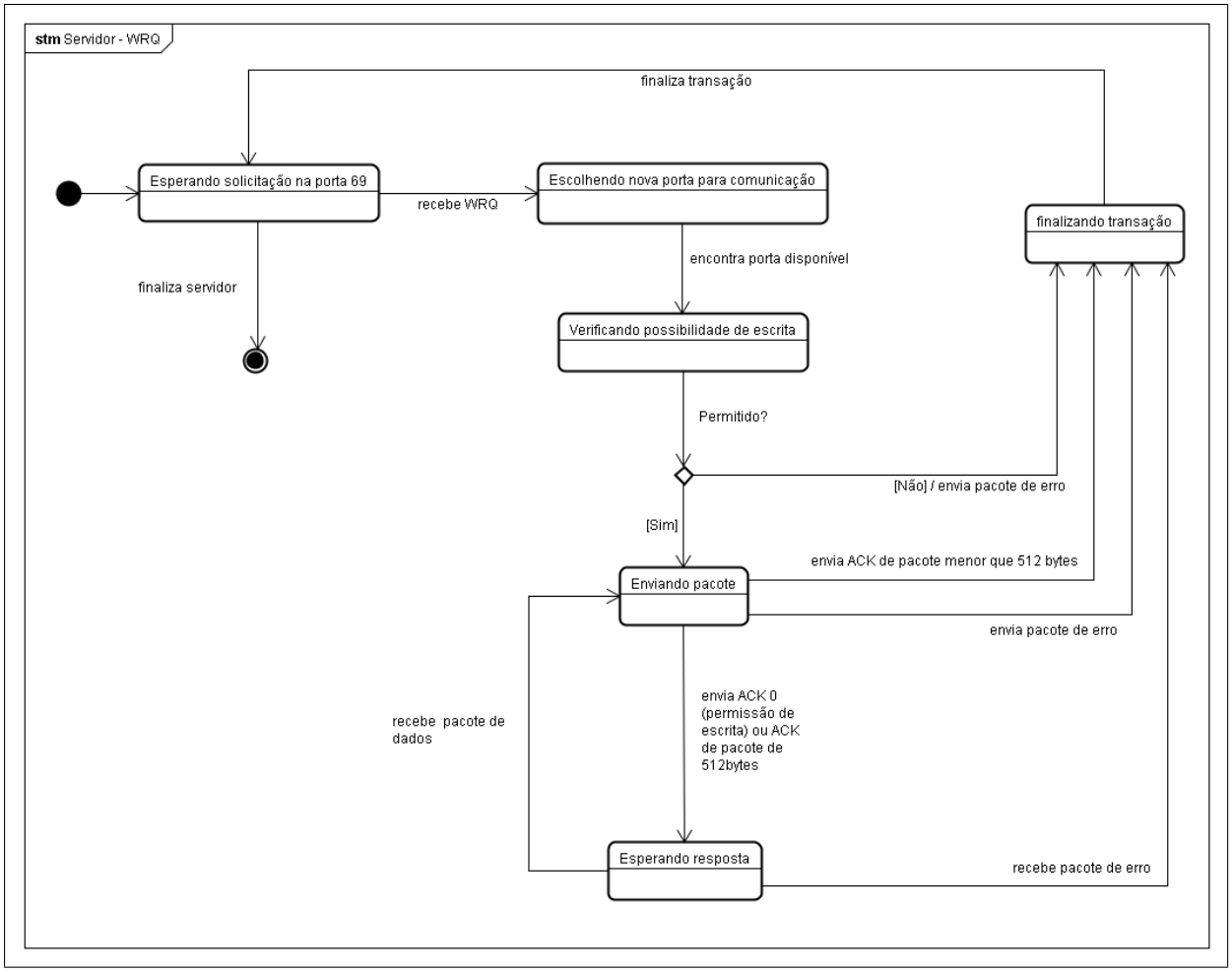

Figura 4.6: Diagrama de Estado - Servidor WRQ

bytes o servidor envia uma confirmação (ack) e passa para o estado *Esperando resposta*, se o pacote de dados recebido for de tamanho menor que 512 bytes o servidor envia a confirmação e passa para o estado *Finalizando transação*. Se o servidor recebe um pacote de erro ele passa para o estado *Finalizando transação*.

Se algum erro ocorreu com o pacote de dados recebido, um pacote de erro é enviada para o cliente e o servidor passa para o estado *Finalizando transação*

4.3 Codificação

O protocolo TFTP foi implementado em dois sistemas diferentes o cliente TFTP e o servidor TFTP.

4.3.1 Cliente TFTP

O software do cliente tem duas funções principais que podem ser chamadas pelo usuário: *cliente_RRQ/2* que solicita a leitura de um arquivo do servidor e *cliente_WRQ/2* que

solicita escrever um arquivo no servidor.

Cliente_RRQ

A função *cliente_RRQ* é mostrado no código 4.3.1, possui dois parâmetros de entrada o ip do servidor *IP* e o nome do arquivo *Arquivo* que será solicitado do servidor.

Na linha 3 do código 4.3.1 um socket UDP é aberto no modo binário, na linha 5 o nome do arquivo é passado para binário para poder ser enviado para o servidor e na linha 6 a solicitação é enviada para o servidor na porta 69. Depois de enviar o pedido para o servidor o cliente chama na linha 7 a função *wait_resposta* na qual espera a resposta do servidor.

```
1  cliente_RRQ(IP,Arquivo)->
2          io:format("\n Solicitando RRQ do servidor",[]),
3          {ok,Socket}=gen_udp:open(5000,[binary]),
4          io:format("\n Socket cliente aberto : ~p ",[Socket]),
5          Arq=list_to_binary(Arquivo),
6          ok = gen_udp:send(Socket,IP,69,<<1:16,Arq/binary,"\0","netascii","\0">>),
7          wait_resposta({Socket,IP,69},0,Arquivo,0,rrq).
```

Código 4.3.1: *Funçao cliente_RRQ*

A função *wait_resposta/5* é mostrada no código 4.3.2 e possui cinco parâmetros de entrada, {*Socket,Server,Porta*} que uma *tuple* contendo o socket, o ip do servidor e a porta para a comunicação, o *NumBloco* que representa o atual número de bloco, *Arquivo* que é o nome do arquivo que vai ser lido, *NumTentativas* que representa o número de tentativvas de enviar o último bloco e um quinto argumento *Dados* que no caso de leitura é o Atom *rrq* e na escrita são os dados do último pacote enviado.

Na linha 2 do código 4.3.2 é mostrado que o bloco esperado (*BlocoEsperado*) é o bloco atual *NumBloco* mais 1, a linha 5 mostra que quando um pacote é recebido os 16 primeiros bits do pacote são colocados na variável *Opcode* e o restando do pacote na variável *Resto*.

```
1  wait_resposta({Socket,Server,Porta},NumBloco,Arquivo,NumTentativas,Dados)->
2     BlocoEsperado=NumBloco+1,
3     receive
4        {udp,Socket,Host,Port,Pacote}->
5           <<Opcode:16,Resto/binary>> = Pacote,
6                    .....
7
8         after 1000->
9                   ......
10    end.
```

Código 4.3.2: *Função wait_resposta*

O cliente interpreta três tipos de *Opcode*: 3 para pacote de dados, 4 para confirmaçao e 5 em caso de pacote de erro. Após identificar o valor do *Opcode* do pacote o cliente durante

uma solicitação de leitura pode entrar em dois trechos de código diferentes: tratamento de pacote de erro e tratamento de pacotes de dados.

Se o *Opcode* é 3, o pacote é de dados e deve ser gravado no cliente como um novo arquivo, o trecho de tratamento de pacote de dados é mostrado no código 4.3.3. Na linha 4 os 16 primeiros bists da variável *Resto* são passados para a variável *Bloco* e o restante para *Data*. Na linha 5 é iniciado um *case* com a variável *Bloco* e com três opções: *Bloco=BlocoEsperado*, Bloco=*NumBloco* e *Bloco=OutroBloco*.

```
1  %***********************************   RRQ   ***********************************
2  %caso seja pacote de dados
3  Opcode==3->
4      <<Bloco:16,Data/binary>> = Resto,
5          case Bloco of
6              BlocoEsperado       ->
7                     Dado=binary_to_list(Data),
8                     io:format("\n Cliente recebeu pacote de dados.
9                                Opcode = ~p, NumBloco = ~p e Dado =\n ~p \n",[Opcode,Bloco,Dado]),
10                    escrever_arquivo(Arquivo,Dado),
11                    PacoteACK = <<4:16,Bloco:16>>,
12                    ok=gen_udp:send(Socket,Host,Port,PacoteACK),
13                    if
14                      byte_size(Data) <512 ->
15                         io:format("\n \n\n       Transmissao terminada com sucesso!!! \n",[]),
16                         io:format("\n Fechando socket \n",[]),
17                         gen_udp:close(Socket);
18                      true->
19                         wait_resposta({Socket,Host,Port},Bloco,Arquivo,1,rrq)
20                    end;
21
22               NumBloco   ->  %recebeu novamente o mesmo bloco
23                    io:format("\n Cliente recebeu pacote duplicado. NumBloco = ~p \n",[NumBloco]),
24                    io:format("\n Cliente envia novamenteACK para o bloco = ~p \n",[NumBloco]),
25                    PacoteACK = <<4:16,Bloco:16>>,
26                    ok=gen_udp:send(Socket,Host,Port,PacoteACK),
27                    wait_resposta({Socket,Host,Port},NumBloco,Arquivo,1,rrq);
28
29               OutroBloco ->
30                    io:format("\nCliente recebeu Ref errado ~p, continua esperando\n",[OutroBloco]),
31                    wait_resposta({Socket,Host,Port},NumBloco,Arquivo,1,rrq)
32
33           end;  %endcase
```

Código 4.3.3: *Função wait_resposta para a leitura de arquivo para Opcode igual a 3*

Caso entre na primeira opção *Bloco=BlocoEsperado*, mostrada entre as linhas 6 e 20, o número de bloco recebido é o esperado, então os dados contidos na variável *Dados* são gravados em um arquivo através da chamada da função *escrever_arquivo* na linha 10. Depois é feito o pacote de confirmação, na linha 11 a variável *PacoteACK* recebe nos 16 primeiros bits o número 4 que significa pacote de confirmação e nos próximos 16 bits é colocado o número do bloco que foi recebido, na linha 12 o pacote é enviado para o servidor. Depois de enviar a confirmação para o servidor, o cliente verifica se o pacote recebido tem tamanho menor que 512 bytes. Se for menor, o socket é fechado na linha 17 finalizando a comunicação, mas se o pacote for de tamanho igual a 512 bytes o cliente chama novamente a função *wait_resposta* na linha 19 passando como número de bloco o bloco atual recebido.

Caso entre na segunda opção *Bloco=NumBloco*, mostrada entre as linhas 22 e 27, o número de bloco recebido é o mesmo atual, ou seja, o bloco foi recebido novamente por que o servidor não recebeu a confirmaçao anterior e enviou o mesmo bloco outra vez. Então o bloco de dados é descartado e uma nova mensagem de confirmação é feita na linha 25, com o mesmo número de bloco e enviada para o servidor na linha 26. O cliente continua esperando o próximo bloco chamando novamente a função *wait_resposta* na linha 27.

Caso entre na terceira opção *Bloco=OutroBloco*, mostrada entre as linhas 29 e 31, o número de bloco recebido no pacote não é o atual nem o próximo, ou seja é um pacote errado e o cliente continua esperando pelo próximo bloco chamando a própria função *wait_resposta* com o mesmo número de bloco.

Cliente_WRQ

A função *cliente_WRQ* é mostrado no código 4.3.4, possui dois parâmetros de entrada o ip do servidor *IP* e o nome do arquivo *Arquivo* que será escrito no servidor.

Na linha 2 do código 4.3.4 o cliente verifica se o arquivo existe, caso não exista é impressa uma mensagem de erro para o usuário e a solicitação não é enviada para o servidor. Se o arquivo existe e pode ser aberto, o cliente abre um socket na linha 5, passa o nome do arquivo para binário e na linha 8 a solicitação é enviada para o servidor na porta 69 com *Opcode* igual a 2. Depois de enviar o pedido para o servidor o cliente chama na linha 9 a função *wait_resposta* na qual espera a resposta do servidor.

```
1  cliente_WRQ(IP,Arquivo)->
2          case file:open(Arquivo,read) of
3                  {ok,ArqPid}->
4                          io:format("\n Solicitando WRQ do servidor",[]),
5                          {ok,Socket}=gen_udp:open(5000,[binary]),
6                          io:format("\n Socket cliente aberto : ~p ",[Socket]),
7                          Arq=list_to_binary(Arquivo),
8                          ok = gen_udp:send(Socket,IP,69,<<2:16,Arq/binary,"\0","netascii","\0">>),
9                          wait_resposta({Socket,IP,69},0,ArqPid,1,wrq);
10                 {error, enoent} ->
11                         io:format("\n\n ERRO: O arquivo não existe~n",[]);
12                 {error, eacces} ->
13                         io:format("\n\n ERRO: Sem permissão para acessar o arquivo ou diretório~n",[]);
14                 {error, eisdir} ->
15                         io:format("\n\n ERRO: O nome especificado não é um arquivo~n",[]);
16                 {error, enotdir} ->
17                         io:format("\n\n ERRO: Um componente do nome do arquivo não é um diretório~n",[])
18         end. %endcase file
```

Código 4.3.4: *Funçao cliente_WRQ*

O código da função *wait_resposta* é mostrado no código 4.3.2, o cliente durante uma solicitação de escrita pode entrar em dois trechos de código diferentes: tratamento de pacote de erro e tratamento de pacotes de confirmação.

Se o *Opcode* é 4 significa que o pacote é de confirmação, deve ser verificado se a confirmação recebida é do último pacote de dados enviado pelo cliente ou de um pacote anterior. Os primeiros 16 bists da variável *Resto* são passados para a variável *Bloco*, e é iniciado um *case* com a variável *Bloco* e com três opções: *Bloco=NumBloco* e *Bloco=BlocoAnterior*.

Caso entre na opção *Bloco=NumBloco* a confirmação recebida é referente ao último pacote de dados enviado para o servidor, essa opção é mostrada no código 4.3.5. Na linha 2 o arquivo é aberto, se ocorrer algum erro durante a leitura do arquivo um pacote de erro é criado na linha 22 e enviado para servidor na linha 23, depois de enviar o erro o socket é fechado na linha 24. Se o arquivo foi abert normalmente, um pacote de dados é criado na linha 15 com número de bloco igual ao atual mais 1 e enviado para o servidor na linha 16, após enviar o pacote de dados é chamda a função *wait_resposta* com o número do novo bloco enviado e os dados enviados como parâmetros.

```
1  NumBloco ->
2          case  file:read(Arquivo,512) of
3             eof->
4                  case  file:close(Arquivo) of
5                          ok->
6                                  io:format("~n Fim do arquivo, arquivo fechado ~n",[]);
7                          {error,Razao}->
8                                  io:format("~n Erro no fechamento do arquivo e razao = ~w ~n",[Razao])
9                  end;
10
11            {ok,Texto}->
12                 io:format("~n~n Texto lido =~p ~n~n",[Texto]),
13                 BlocoNovo=Bloco+1,
14                 Dado=list_to_binary(Texto),
15                 PacoteDado = <<3:16,BlocoNovo:16,Dado/binary>>,
16                 ok=gen_udp:send(Socket,Host,Port,PacoteDado),
17                 wait_resposta({Socket,Host,Port},BlocoNovo,Arquivo,1,Texto);
18
19             {error,Razao}->
20                 io:format("~n Erro na leitura e razao = ~w ~n",[Razao]),
21                 ErrMsg=term_to_binary(Razao),
22                 PacoteErro = <<5:16,0:16,ErrMsg/binary>>,
23                 ok=gen_udp:send(Socket,Host,Port,PacoteErro),
24                 gen_udp:close(Socket)
25
26
27         end;
```

Código 4.3.5: *Função wait_resposta - recebe confirmação de pacote atual*

Caso entre na opção *Bloco=BlocoAnterior* a confirmação recebida é referente a um pacote de dados anteriormente enviado para o servidor, ou seja, o servidor não recebeu o novo pacote de dados e reenviou a confirmaçao anterior, essa opção é mostrada no código 4.3.6. Na linha 4 o pacote de dados é feito com os dados enviados anteriormente e não recebidos pelo servidor, na linha 5 é enviado o pacote. A função *wait_resposta* é chamada com o mesmo número de bloco anterior.

```
1  BlocoAnterior -> %recebe confirmação do bloco anterior
2          io:format("~n~n Recebeu ACK ~p denovo, reenviando pacote de dados ~n~n",[Bloco]),
3          Dado=list_to_binary(Dados),
4          PacoteDado = <<3:16,NumBloco:16,Dado/binary>>,
5          ok=gen_udp:send(Socket,Host,Port,PacoteDado),
6          wait_resposta({Socket,Host,Port},NumBloco,Arquivo,1,Dados);
```

Código 4.3.6: *Função wait_resposta - recebe confirmação de pacote anterior*

4.3.2 Servidor TFTP

O software do servidor tem uma função principal que inicia o funcionamento do servidor,*start_servidor/0* que abre um socket UDP na porta 69 e chama a função *servidor/1* que fica esperando solicitações de clientes. A função *start_servidor* é mostrada no código 4.3.7.

```
1  start_servidor()->
2          {ok,Socket}=gen_udp:open(69,[binary]),
3          io:format("\nServidor UDP com Ref aberto socket : ~p~n",[Socket]),
4          servidor(Socket).
```

Código 4.3.7: *Função start_servidor*

A função *servidor/1* que é mostrada no código 4.3.8 fica escutando na porta 69, esperando solicitações de clientes.Quando um pacote chega, seu *Opcode* é extraido e verificado se é uma solicitação de leitura (*Opcode* 1) ou escrita (*Opcode* 2). Então um novo processo é criado para lidar com essa requisição.

```
1  servidor(Socket)->
2      receive
3          {udp,Socket,Host,Port,Pacote}->
4              <<Opcode:16,Resto/binary>> = Pacote,
5                  ListaResto=binary_to_list(Resto),
6                  ListaParametro=parse:parser(ListaResto,[],[]),
7                  [Arquivo|Modo]=ListaParametro,
8                  if
9                      Opcode==1-> %solicitação rrq-leitura de arquivo
10                         io:format("\n Servidor recebeu solicitacao de leitura ~n",[]),
11                         spawn(fun() -> loop_leitura(Host,Port,Arquivo) end),
12
13                     Opcode==2->%solicitação wrq-escrita de arquivo
14                         io:format("\n Servidor recebeu solicitacao de escrita ~n",[]),
15                         spawn(fun() -> loop_escrita(Host,Port,Arquivo) end),
16
17                     true->
18                         io:format("\n Opcode indefinido: ~p~n",[Opcode]),
19
20                 end
21
22     end,
23     servidor(Socket).
```

Código 4.3.8: *Função servidor*

Servidor_RRQ

Caso o *Opcode* seja 1 (uma solicitação de leitura), o novo processo criado é chamado com a função *loop_leitura* que lê o arquivo no servidor e envia para o cliente. Na função é verificado se o arquivo solicitado existe e se pode ser enviado.

Caso o arquivo não exista o trecho mostrado no código 4.3.9 é executado, um pacote de erro é criado na linha 2 e a mensagem é enviada para o cliente na linha 4. Após enviar o erro a comunicação é finalizada.

```
1  io:format("~n ERRO arquivo = ~w nao encontrado, Razao = ~w ~n~n",[Arquivo,Razao]),
2  ErrMsg=term_to_binary(Razao),
3  PacoteErro = <<5:16,1:16,ErrMsg/binary>>,
4  ok=gen_udp:send(Socket,Host,Port,PacoteErro)
```

Código 4.3.9: *Função loop_leitura - erro de arquivo inexistente*

Caso o arquivo exista, o primeiro bloco de dados de 512 bytes lido é enviado para o cliente, esse trecho é mostrado no código 4.3.10. Na linha 1 os dados lidos do arquivo são passados para binário para ser enviado ao servidor, na linha 2 o pacote de dados é criado com *Opcode* igual a 3 e número de bloco igual a 1 por ser o primeiro bloco de dados e na linha 3 o pacote é enviado para o cliente. Na linha 4 a função *leitura/5* é chamada para esperar as confirmações do cliente e enviar os próximos blocos de dados.

```
1  Dado=list_to_binary(Texto),
2  PacoteDado = <<3:16,1:16,Dado/binary>>,
3  ok=gen_udp:send(Socket,Host,Port,PacoteDado),
4  leitura({Socket,Host,Port},1,ArqPid,1,Texto)
```

Código 4.3.10: *Função loop_leitura*

A função *leitura/5* possui cinco parâmetros de entrada, {*Socket,Client,Porta*} que é uma *tuple* contendo o socket, ip do cliente e porta da comunicação, *NumBloco* que representa o atual número de bloco, *Arquivo* que é o nome do arquivo que vai ser lido, *NumTentativas* que representa o número de tentativas de enviar o pacote e *Dados* que são os dados enviados no último pacote. Esta função espera o pacote de confirmaçao enviado pelo cliente, caso a confirmação recebida seja referente ao pacote enviado anteriormente o servidor empacota mais 512 bytes do arquivo e envia para o cliente, caso a confirmação seja do pacote anterior o servidor envia novamente esse pacote.

Servidor_WRQ

Caso o *Opcode* seja 2 (uma solicitação de escrita), o novo processo criado é chamado com a função *loop_escrita* que verifica a possibilidade de escrita do arquivo e manda o

pacote de permissão de escrita para o cliente.

Parte da função *loop_escrita* é mostrada no código 4.3.11, na linha 1 é criado o pacote de permissão no qual os 16 primeiros bits são o código 4 de confirmação e os próximos 16 bits são 0 que significa permissão de escrita. Na linha 2 o pacote é enviado para o cliente e na linha 3 a função *escrita/4* é chamada, com o segundo parâmetro igual a 0 representando o pacote de permissão. Na função *escrita/4* o servidor espera pelos pacotes de dados do cliente e os responde com pacotes de confirmação.

```
1  PacoteACK = <<4:16,0:16>>,
2  ok=gen_udp:send(Socket,Host,Port,PacoteACK),
3  escrita({Socket,Host,Port},0,Arquivo,1)
```

Código 4.3.11: *Função loop_escrita*

A função *escrita/4* possui quatro parâmetros: {*Socket,Client,Porta*} que é uma *tuple* que representa o socket,ip e porta para a comunicação, *NumBloco* que representa o atual número de bloco, *Arquivo* que é o nome do arquivo que vai ser escrito no servidor e *Num-Tentativas* que é o número de tentativas de enviar o pacote. A função espera pelo bloco de número igual *NumBloco* mais 1, se este bloco é recebido o servidor escreve no arquivo os dados recebidos no pacote e envia um pacote de confirmação para o cliente, como mostrado no código 4.3.12

```
1  {udp,Socket,Host,Port,Pacote}->
2          <<Opcode:16,Resto/binary>> = Pacote,
3                  .....
4
5          <<Bloco:16,Data/binary>> = Resto,
6                  .....
7
8          Dado=binary_to_list(Data),
9          escrever_arquivo(Arquivo,Dado),
10         PacoteACK = <<4:16,Bloco:16>>,
11         ok=gen_udp:send(Socket,Host,Port,PacoteACK),
12         if
13                 byte_size(Data) <512 ->
14                         io:format("\n Fechando socket \n",[]),
15                         gen_udp:close(Socket);
16                 true->
17                         escrita({Socket,Host,Port},Bloco,Arquivo,1)
18         end;
```

Código 4.3.12: *Função escrita quando recebe a confirmação do bloco atual*

Caso a confirmação recebida seja de um pacote mandado anteriomente, o servidor empacota novamente os últimos dados lidos do arquivo com o mesmo número de bloco e reenvia o pacote para o cliente.

Capítulo 5
Conclusão

O Desenvolvimento do protocolo TFTP na liguagem funcional Erlang mostrou ser possível aprender a utilizar os mecânimos da linguagem e comprovar que o Erlang é aplicável ao desenvolvimento de protocolos. Os objetivos estipulados foram atendidos, as aplicações cliente e servidor foram desenvolvidas e conseguem se comunicar entre si e com aplicações TFTP externas.

Diferente de outras linguagens o Erlang foi criado para resolver um problema bem definido, ser tolerante a falhas, lidar com concorrência pesada, usar computação distribuída, atualizar aplicações sem derrubá-las, criar sistemas de tempo real. Por isso muitas vezes é visto como uma linguagem difícil que gera códigos complexos, mas o Erlang é uma linguagem declarativa que facilita a escrita de código simples, objetivo, coeso e modular. Cria código fácil de ler, escrever e organizar em módulos reutilizáveis.

O protocolo TFTP é uma boa opção para transferência de arquivos quando não se necessita de precisão na entrega dos pacotes, não é necessário uma visualização dos diretórios nem a autenticação do usuário que está acessando o servidor TFTP. Por ser um protocolo simples de aprender e implementar, um software TFTP poderia ser utilizado em um pequena empresa para armazenamento e transferência de arquivos em uma intranet, sem a necessidade de utilizar softwares proprietários.

5.1 Trabalhos futuros

Para complementar o trabalho desenvolvido são sugeridos para trabalhos futuros:

- Criação das interfaces gráficas para cliente e servidor - para que os usuários não tenham que lidar com a linha de comando.
- Criação de um instalador - para que usuários possam em simples passos ter a aplicação em seus computadores.

Referências Bibliográficas

[Armstrong2007] Armstrong, J. (2007). *The Pragmatic Programmers Programming Erlang*. Copyright.

[Armstrong et al.2007] Armstrong, J., Virding, R., Wikstrom, C., and Williams, M. (2007). *Concurrent Programming in ERLANG*. PRENTICE HALL, second edition.

[da Silva Vieira2011] da Silva Vieira, L. I. (2011). Desenvolvimento de um jogo multiusuário online usando programaçao concorrente.

[Ericsson] Ericsson. Getting started with erlang user's guide.

[Fairhurst] Fairhurst, G. The user datagram protocol (udp).

[Gavidia] Gavidia, J. J. Z. Tftp (trivial file transfer protocol).

[TechNet] TechNet, M. Visão geral do tftp (trivial file transfer protocol).

[Williams] Williams, A. *JAVA 2 Network Protocols. Black Book*. Steve Sayre.

Capítulo 6
Apêndice

O código fonte do cliente e servidor TFTP está no CD em anexo. O cliente é o arquivo *cliente_tftp.erl* e o servidor é o arquivo *servidor_tftp.erl*.