INSTITUTO DO EMPREGO
E FORMAÇÃO PROFISSIONAL

CET 06 - Técnico Especialista em Gestão de Redes e Sistemas Informáticos

Python – Projecto 2
(TFTPy – Cliente / Servidor)

Feito por: **Pedro Pereira** e **Pedro Lourenço** (11/06/2017)

UFCD 5118 – Programação

Conteúdo

Introdução e Objectivos

Introdução

Este projeto tem como objetivo permitir ficar com um melhor conhecimento sobre o funcionamento do protocolo TFTP, sobre o módulo **socket** e ao mesmo tempo reforçar o conhecimento da biblioteca/módulo **argparse** ou **docopt** adquirido no projecto anterior.

O que é o protocolo TFTP?

O protocolo **TFTP** (Trivial File Transfer Protocol) é um protocolo simples que permite a transferência de ficheiros entre máquinas (semelhante ao protocolo FTP). Devido à popularidade do **VoIP** (Voz sobre IP), o protocolo TFTP tornou-se um protocolo muito procurado, porque permite aprovisionar telefones VoIP de forma simplificada, isto é, enviar de forma automática a configuração para os terminais VoIP.

É igualmente utilizado para a actualização do firmware em equipamentos.

Este protocolo utiliza a porta 69 (por default) e é baseado em UDP, não contendo suporte nativo para mecanismos de autenticação e encriptação dos dados. Suporta três modos de transferência (netascii, octet ou mail).

Para que serve o módulo socket?

O módulo socket permite que programas (processos) estabeleçam comunicação entre si através de um canal de comunicação na rede (bidirecional), baseando-se no modelo cliente/servidor, onde os dados são divididos em blocos (por ex: de 512 bytes) e enviados através do socket, sendo reconstruídos no outro lado (receptor).

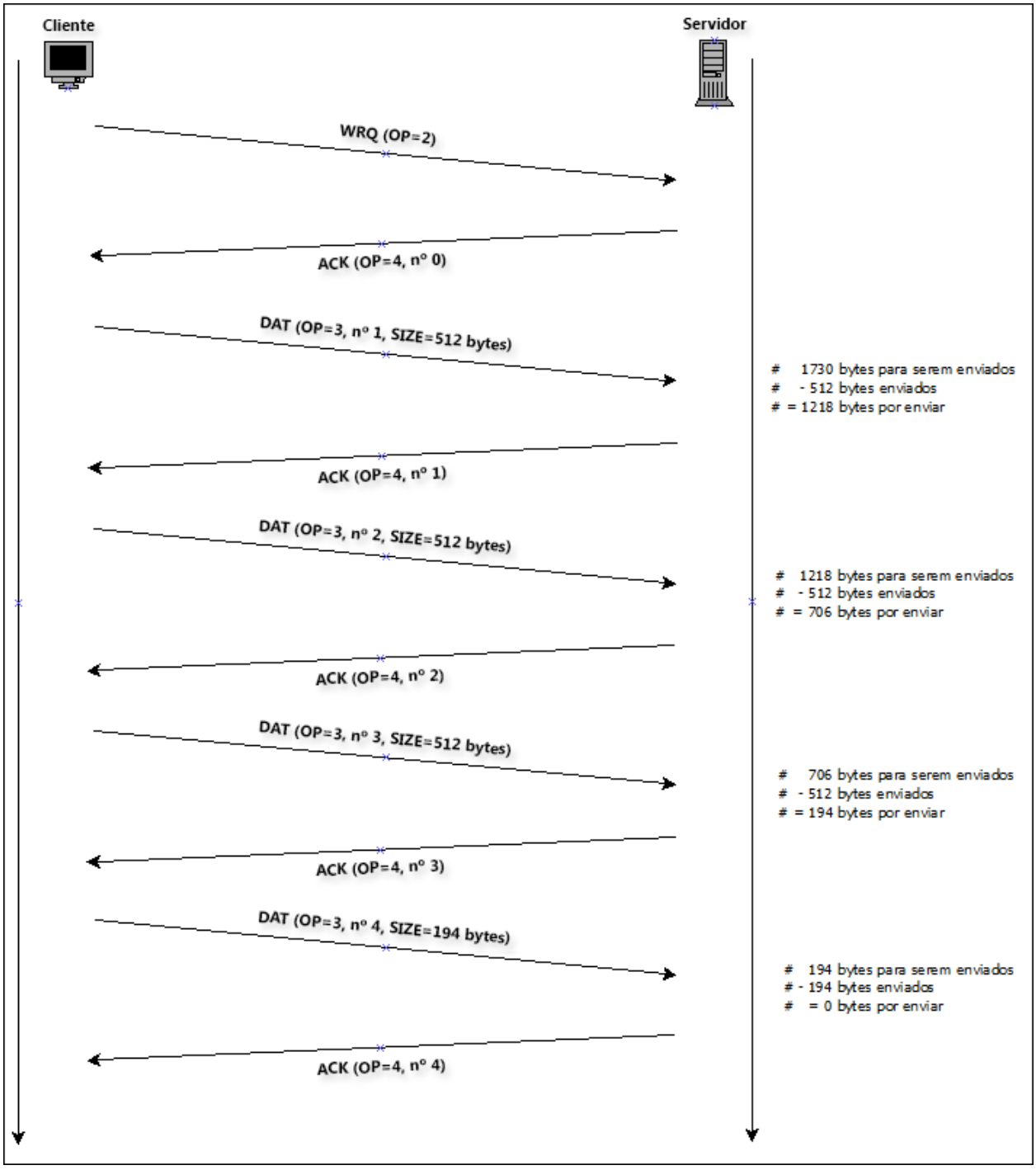
Objectivos

- Desenvolver uma aplicação **cliente** que permita fazer transferências de ficheiros, baseado no protocolo TFTP;
- Desenvolver uma aplicação **servidor** que permita fazer transferências de ficheiros, baseado no protocolo TFTP (opcional);
- No caso de se ter desenvolvido a aplicação **servidor**, adicionar um mecanismo de segurança via TLS/SSL (opcional), adicionar o comando dir para que se possa listar o conteúdo da aplicação **servidor** (opcional) e adicionar um temporizador de reenvio de ACK / DAT (opcional).

Nota: Devido a alguns problemas de última hora no funcionamento dos programas, que inclusive, motivou o atraso de um dia no envio do projecto, foi necessário fazer as devidas correções / alterações no código desenvolvido. Esses problemas foram ultrapassados, mas devido ao tempo ser curto (próximo da meia-noite) e não querendo levar com uma penalização maior, não foi possível actualizar a descrição feita na secção implementação. Assim sendo a mesma encontra-se um pouco desactualizada em relação ao código final dos programas.

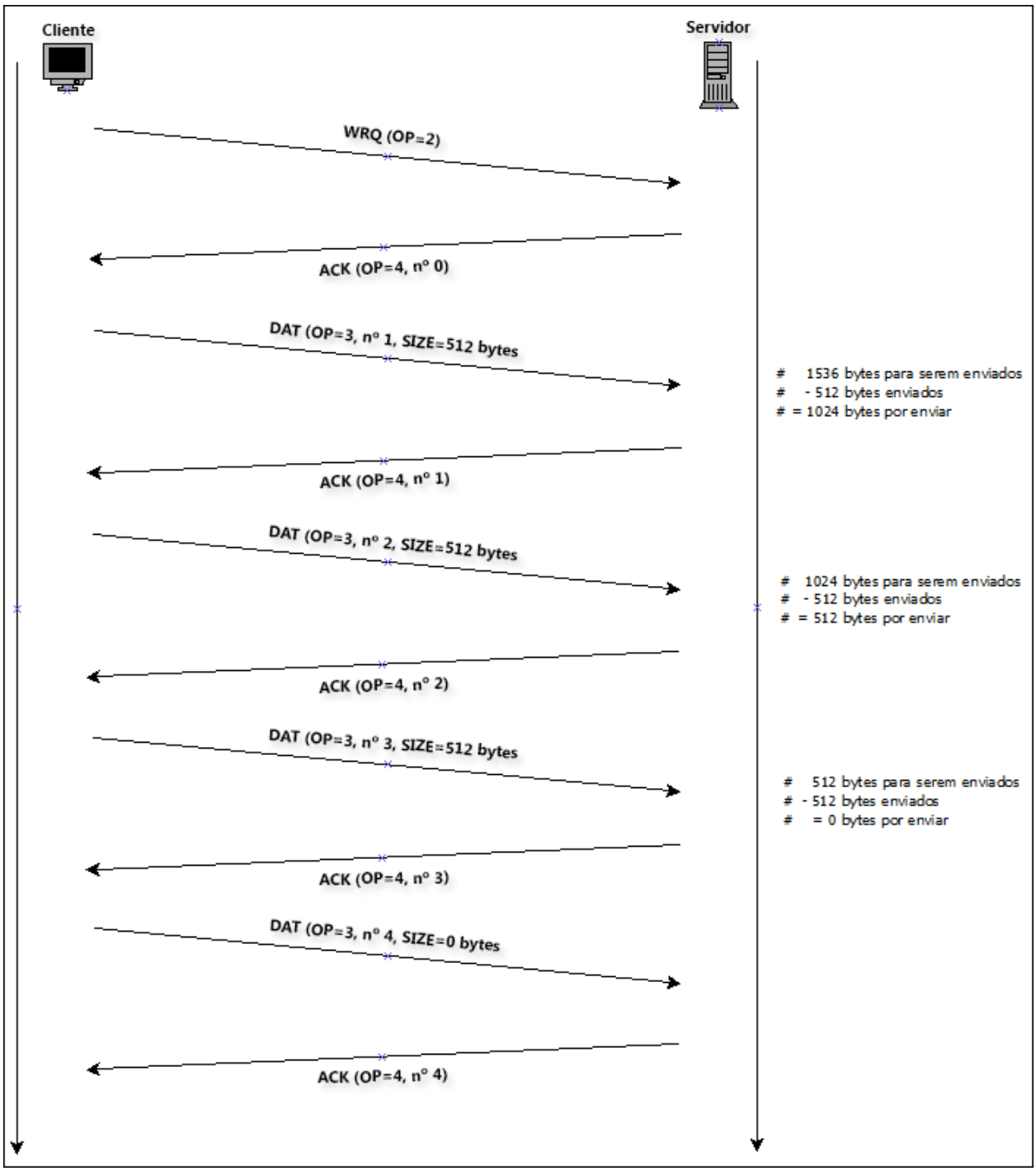
Análise

Diagrama de mensagens a ilustrar o envio de um ficheiro com 1730 bytes



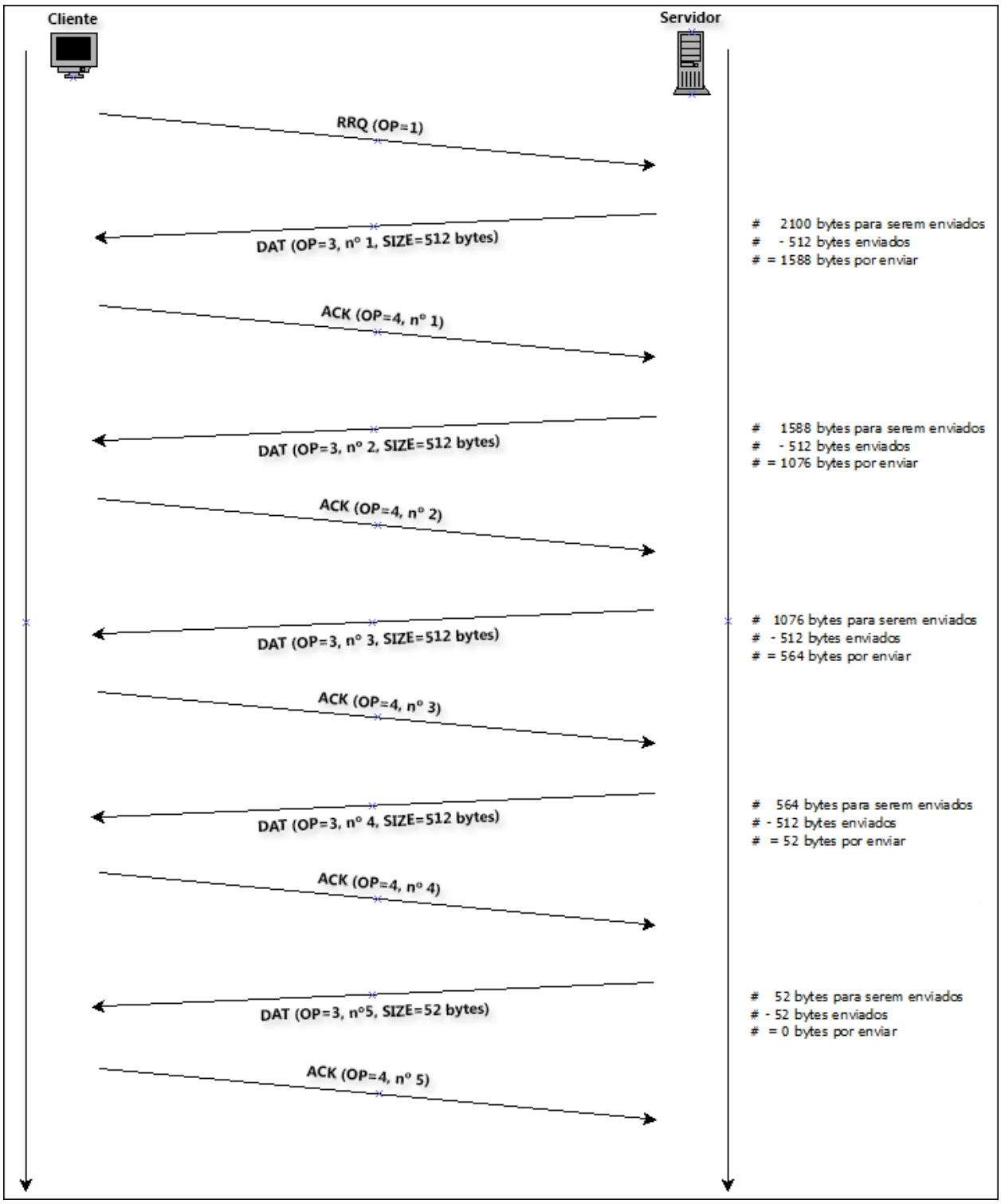
Nota: Uma vez que a imagem do diagrama perde alguma definição aqui no Word e que limita um pouco a sua leitura em condições, no ficheiro .zip do projecto vai incluída a imagem do diagrama em causa, que permite a sua devida leitura.

Diagrama de mensagens a ilustrar o envio de um ficheiro com 1536 bytes



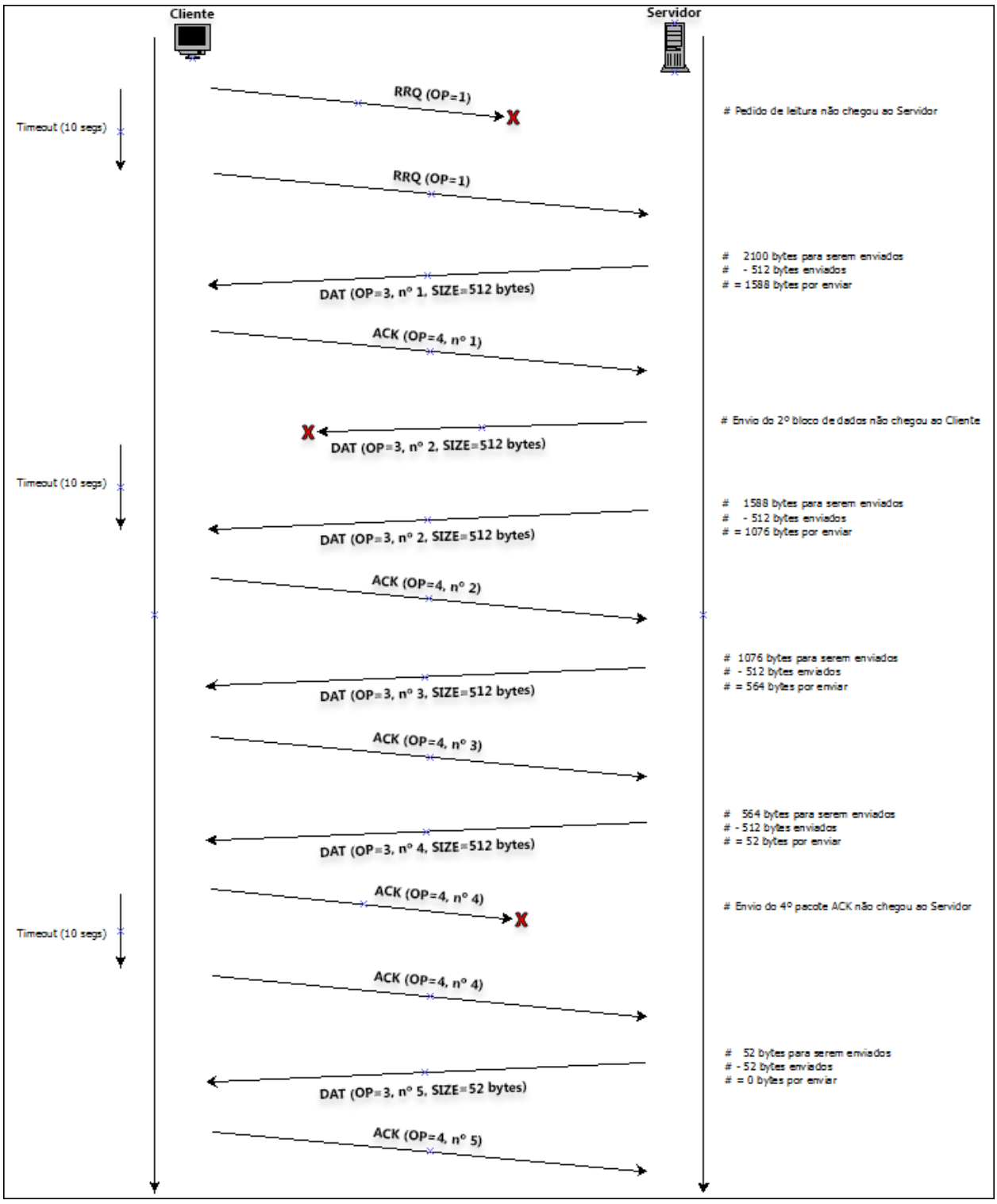
Nota: Uma vez que a imagem do diagrama perde alguma definição aqui no Word e que limita um pouco a sua leitura em condições, no ficheiro .zip do projecto vai incluída a imagem do diagrama em causa, que permite a sua devida leitura.

Diagrama de mensagens a ilustrar a recepção de um ficheiro com 2100 bytes



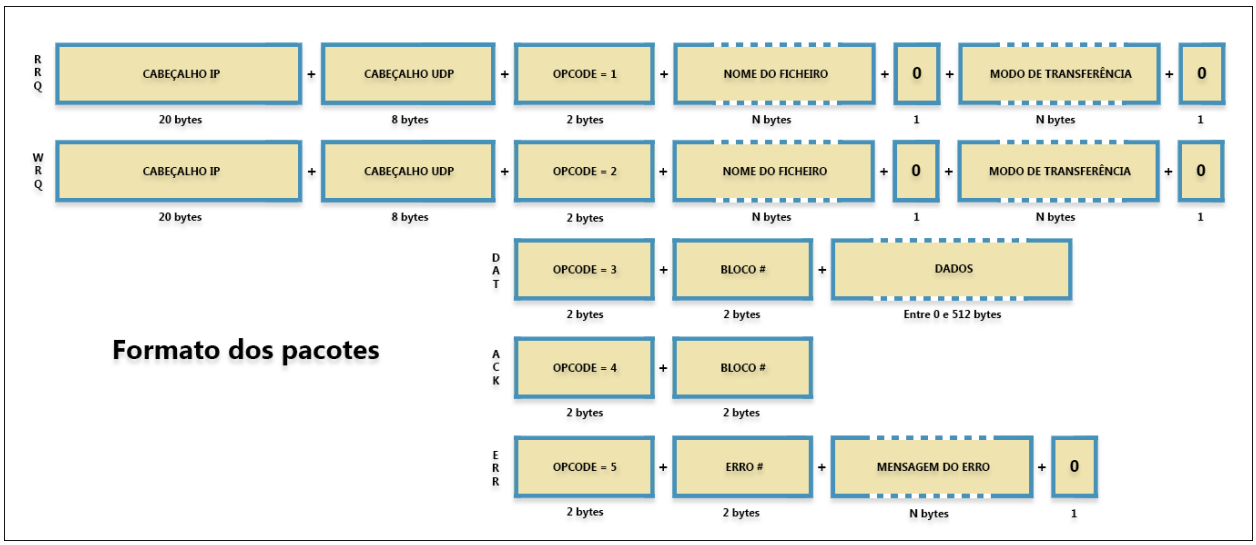
Nota: Uma vez que a imagem do diagrama perde alguma definição aqui no Word e que limita um pouco a sua leitura em condições, no ficheiro .zip do projecto vai incluída a imagem do diagrama em causa, que permite a sua devida leitura.

Diagrama de mensagens a ilustrar a recepção de um ficheiro, contemplando falhas/erros



Nota: Uma vez que a imagem do diagrama perde alguma definição aqui no Word e que limita um pouco a sua leitura em condições, no ficheiro .zip do projecto vai incluída a imagem do diagrama em causa, que permite a sua devida leitura.

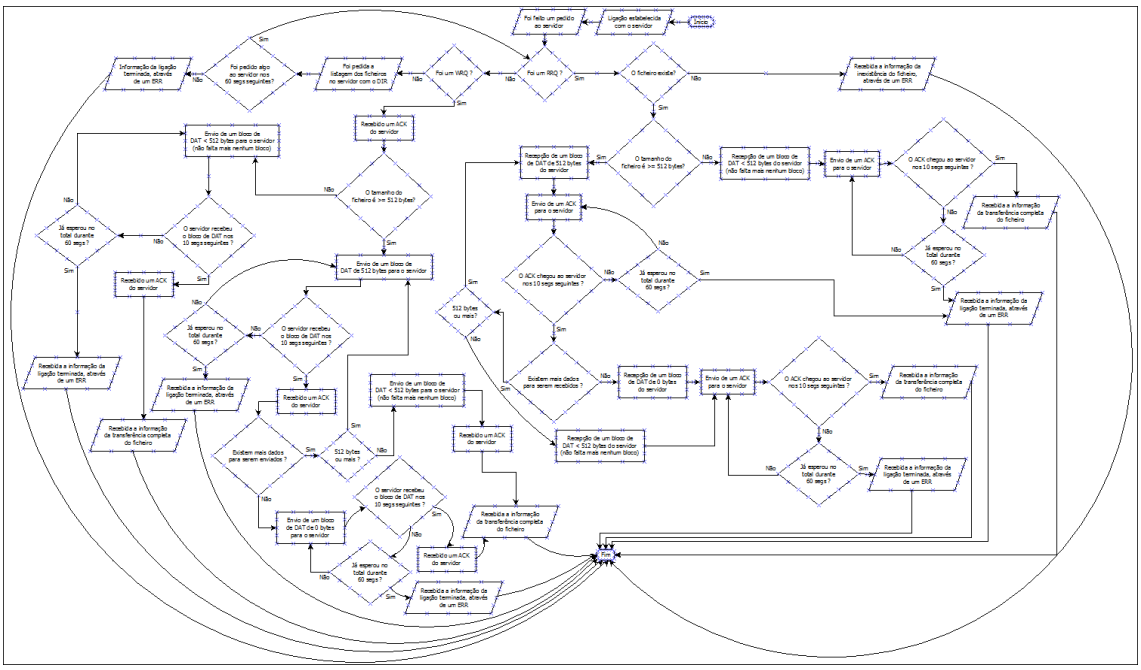
Diagrama com o formato dos pacotes



Nota: Uma vez que o diagrama é grande e aqui no Word não permite a sua leitura em condições (a não ser que faça zoom in), no ficheiro .zip do projecto vai incluído uma imagem do diagrama em causa que permite a sua devida leitura.

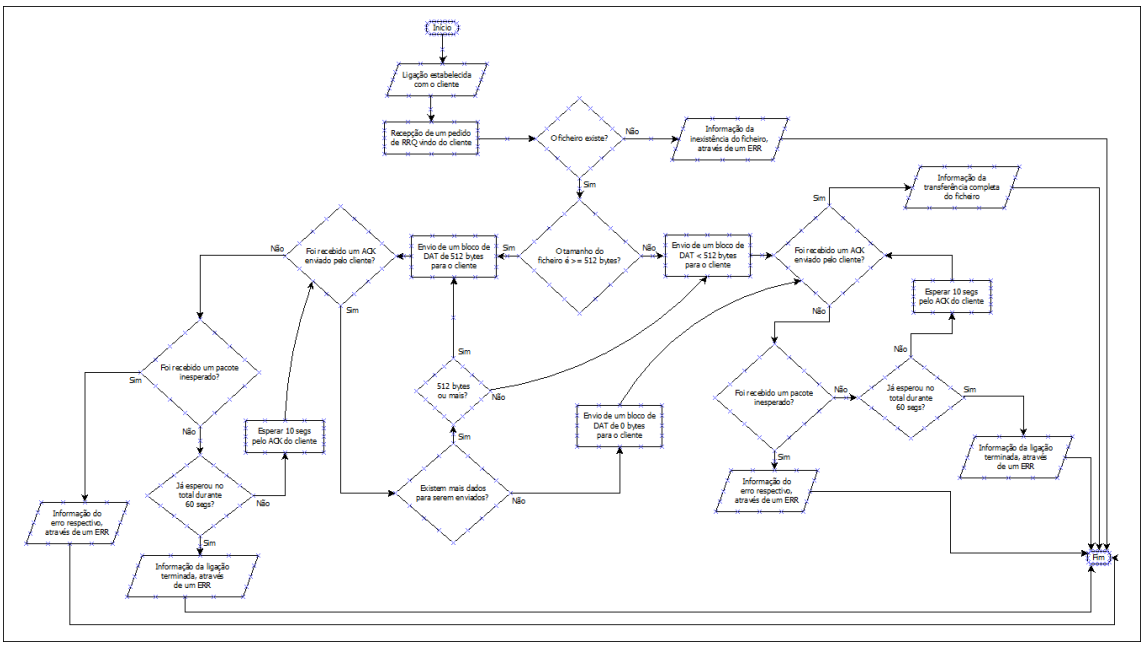
Desenho e Estrutura

Fluxograma - Cliente



Nota: Uma vez que o fluxograma é grande e aqui no Word não permite a sua leitura em condições (a não ser que faça zoom in), no ficheiro .zip do projecto vai incluído uma imagem do fluxograma em causa que permite a sua devida leitura.

Fluxograma - Servidor



Nota: Uma vez que o fluxograma é grande e aqui no Word não permite a sua leitura em condições (a não ser que faça zoom in), no ficheiro .zip do projecto vai incluído uma imagem do fluxograma em causa que permite a sua devida leitura.

Implementação

TFTP.py

Neste ficheiro colocámos todas as funções que desenvolvemos, mais especificamente:

-» def **chunks**

Esta função geradora vai produzir blocos com o tamanho de n bytes do ficheiro file; a instrução yield devolve um valor, até que não haja mais nenhum.

```
def chunks(file, n):
    "Yield successive n-sized chunks from read file"
    blk = 0
    for i in range(0, len(file), n):
        blk+=1
        yield file[i:i+n]
        yield blk
```

-» def **pack_R_W**

Esta função vai construir o pacote de um RRQ / WRQ.

A variável fmt fica com a informação do formato do pacote.

A variável packet_fmt fica com o pacote RRQ / WRQ que foi criado com a função *pack* do módulo *struct*; pacote esse que possui a informação do formato do mesmo, o option code respectivo (RRQ: 1 ou WRQ: 2), o nome do ficheiro e o modo de transferência. Esta variável é devolvida no fim pela função.

```
def pack_R_W(op, filename, mode):
    "Constrói pacote(s) RRQ e WRQ"
    fmt = '!H{}s6s'.format(len(filename), len(mode))
    packet_fmt = struct.pack(fmt, op, filename, mode)
    return packet_fmt
```

-» def **pack_3_**

Esta função constrói um pacote do tipo DAT.

Verifica o tipo de dados do bloco de dados do ficheiro (dat) que será enviado para o receptor, se for string, será codificado em bytes com a função *encode*. Posteriormente é adicionado um “0” (em bytes) no final desse bloco. Se o mesmo já estava em bytes apenas é adicionado o “0” (em bytes) no fim do bloco.

A variável fmt fica com a informação do formato do pacote.

A variável packet_fmt fica com o pacote DAT que foi criado com a função *pack* do módulo *struct*; pacote esse que possui a informação do formato do mesmo, o option code respectivo (DAT: 3), o número do bloco (DAT n) e finalmente o bloco de dados do ficheiro a ser enviado. Esta variável é devolvida no fim pela função.

```
def pack_3_(op, blk, dat):
    "Constrói pack tipo DAT"
    "Formato: op(2bytes) <> blk(2bytes) <> dat(n-bytes)"
    op = 3
    if type(dat) == str:
        dat = dat.encode()
        dat += b'\0'
    else:
        dat += b'\0'
    fmt = '!HH{}s'.format(len(dat))
    packet_fmt = struct.pack(fmt, op, blk, dat)
    return packet_fmt
```

-» def **pack_4_**

Esta função constrói um pacote do tipo ACK.

A variável fmt fica com a informação do formato do pacote.

A variável packet_fmt fica com o pacote ACK que foi criado com a função *pack* do módulo *struct*; pacote esse que possui a informação do formato do mesmo, o option code respectivo (ACK: 4) e o número do bloco (ACK n) a ser enviado. Esta variável é devolvida no fim pela função.

```
def pack_4_(op, blk):
    "Constrói pack tipo ACK"
    "Formato: op(2bytes) <> blk(2bytes)"
    op = 4
    fmt = '!HH'.format()
    packet_fmt = struct.pack(fmt, op, blk)
    return packet_fmt
```

-» def **pack_err**

Esta função constrói um pacote do tipo ERR.

A variável fmt fica com a informação do formato do pacote. A variável packet_fmt fica com o pacote ERR que foi criado com a função *pack* do módulo *struct*; pacote esse que possui a informação do formato do mesmo, o option code respectivo (ERR: 5), o número do bloco (ERR n) e finalmente a mensagem de erro específica a ser enviada. Esta variável é devolvida no fim pela função.

```
def pack_err(op, err, msg):
    "Constrói pack tipo ERR"
    "Formato: op(2bytes) <> err(2bytes) <> err_msg(n-bytes)"
    if isinstance(msg, list):
        msg = msg[0]
        if isinstance(msg, str):
            msg = msg.encode()
    if isinstance(msg, str):
        msg = msg.encode()
    msg += b'\0'
    op = 5
    fmt = '!HH{}s'.format(len(msg))
    packet_fmt = struct.pack(fmt, op, err, msg)
    return packet_fmt
```

-» def **unpack_err**

Esta função vai descompactar um pacote do tipo ERR.

O pacote ERR (pack) será descompactado pela função *unpack* do módulo *struct*, tendo em conta o formato do pacote (fmt); as variáveis op, err e msg recebem os respectivos valores após o *unpac*k, sendo que no caso da variável msg, que contem a mensagem do erro em bytes, será feita a sua descodificação com a função *decode*, sendo igualmente separado o caractere “0” com a função *strip*.

As três variáveis referidas em cima, serão enviadas para um tuple chamado lista, que será devolvido no fim pela função.

```
def unpack_err(pack):
    "Descompacta pack do tipo ERR"
    msg = pack[4:]
    fmt = '!HH{}s'.format(len(msg))
    op, err, msg = struct.unpack(fmt, pack)
    msg = msg.decode().strip('\x00')
    lista = op, err, msg
    return lista
```

-» def **unpack_R_W**

Esta função vai descompactar um pacote do tipo RRQ / WRQ.

O pacote (pack) será descompactado pela função *unpack* do módulo *struct*, tendo em conta o formato do pacote (fmt); as variáveis opcode, fileS e modeS recebem os respectivos valores após o unpack. As variáveis fileS e modeS são descodificadas com a função *decode* com o codec UTF-8. De seguida, é feito um *strip* à variável fileS e enviado para a variável fileDES à qual será feito o *split* do caractere “o”, sendo enviada para a variável fileDesF.

Após isto é verificado o tamanho (número de itens) no objecto fileDesF, isto é, se for 2 é porque o utilizador inseriu um nome específico para dar ao ficheiro original do emissor para o receptor, sendo criadas mais duas variáveis (f_rqq e f_toWrite), uma receberá o nome original do ficheiro e a outra receberá o “novo” nome desse ficheiro no receptor. Depois é feito um *strip* na variável que contem a informação do modo de transferência (modeDES). De seguida, as variáveis opcode, f_rqq, f_toWrite e modeDES são colocadas num tuple com o nome lista, sendo este mesmo devolvido pela função.

Se o *len* do fileDesF não for 2, quer dizer que o utilizador não especificou um nome para o ficheiro original do emissor para o receptor. Sendo assim, só é feito o *strip* ao modeDES, sendo as variáveis opcode, fileDesF e modeDES enviadas de seguida para o tuple lista, que é devolvido pela função.

```
def unpack_R_W(pack):
    "Descompacta 2 tipos de pacotes (RRQ / WRQ)"
    file = pack[2:-6]  # File received
    mode = pack[-6:]  # Mode received
    opcode = pack[:2]  # Opcode received
    fmt = '!H{}s6s'.format(len(file), len(mode))
    opcode, fileS, mode = struct.unpack(fmt, pack)
    fileS = file.decode('utf-8')  # File decoded from bytes
    modeS = mode.decode('utf-8')  # Mode decoded from bytes
    stripped = '\x00-\x01-\x02-\x03-\x04'
    fileDES = fileS.strip(stripped)
    fileDesF = fileDES.split('\x00')
    if len(fileDesF) == 2:
        f_rqq = fileDesF[0]
        f_toWrite = fileDesF[1]
        modeDES = modeS.strip(stripped)
        lista = opcode, f_rqq, f_toWrite, modeDES
        return lista
    else:
        modeDES = modeS.strip(stripped)
        lista = opcode, fileDesF, modeDES
        return lista
```

-» def **unpack_dat**

Esta função vai descompactar um pacote do tipo DAT.

Primeiro é utilizada a função *compile* do módulo *re* para compilar um padrão de uma expressão regular para o objecto pat, depois é utilizada a função *search* do módulo *re* que procura por esse padrão (contido em pat) no pacote recebido (pack), sendo enviado para o objecto chk_type. Finalmente é usada o método *group* que agrupa numa string os resultados onde foi verificado o padrão da expressão regular, sendo enviado para a variável verify.

De seguida é verificado se em verify está o “3” em binário, uma vez que se trata de um pacote de DAT, está condição é **True** e a função passa à descompactação do pacote tendo em conta o formato do empacotamento do pacote, que está presente em fmt, sendo que as variáveis op, blk e data1 recebem os respectivos dados. É retirado o caractere “0” do data1 com o *strip* e o conteúdo enviado para data_c. Finalmente, as variáveis op, blk e data_c são colocadas num tuple com o nome de lista e o mesmo é devolvido no fim pela função.

```
def unpack_dat(pack):
    "Descompacta pack do tipo DAT"
    pat = re.compile(b'(^[\x00-\x05]{1,2})')
    chk_type = re.search(pat,pack)
    verify = chk_type.group()
    if verify == b'\x00\x03':
        data = pack[4:]
        #data_size=len(data)
        fmt = '!HH{}s'.format(len(data))
        op,blk,datal = struct.unpack(fmt,pack)
        data_con = datal.strip(b'\x00')
        data_c = data_con.decode('utf-8', 'ignore')
        lista = op, blk, data_c
        return lista
    else:
        print('Error on unpack DAT')
```

-» def **check_pack**

Esta função determina o tipo de pacote.

Primeiro é utilizada a função *compile* do módulo *re* para compilar um padrão de uma expressão regular para o objecto pat_all, depois é utilizada a função *search* do módulo *re* que procura por esse padrão (contido em pat_all) no pacote recebido (data), sendo enviado para o objecto check. Consoante se for um RRQ, WRQ, etc…, será mostrada a respectiva mensagem.

```
def check_pack(data):
    "Determina o tipo de pacote: RRQ <> WRQ <> DAT <> ACK <> ERR"
    RRQ = b'\x00\x01'
    WRQ = b'\x00\x02'
    DAT = b'\x00\x03'
    ACK = b'\x00\x04'
    ERR = b'\x00\x05'
    DIR = b'\x00\x06'

    pat_all = re.compile(b"(^[\x00-\x05].)")
    chek = re.search(pat_all, data)
    if chek == None:
        return data
    else:
        check = chek.group()
    if check == RRQ:
        out = print("Packet received RRQ: ", check)
    if check == WRQ:
        out = print("Packet received WRQ: ", check)
    if check == DAT:
        out = print("Packet received DAT: ", check)
    if check == ACK:
        out = print("Packet received ACK: ", check)
    if check == ERR:
        out = print("Packet received ERR: ", check)
    if check == DIR:
        out = print("Treating DIR")
    return check
```

-» def **file_verify**

Esta função faz a verificação da existência ou não do ficheiro (file_to_chk).

Se existir (True) é feito o *open* com o modo rb (leitura em bytes) e enviado para f1_, de seguida é feito o *read* para ler o conteúdo do f1_ e enviado para f_. É depois verificado o tamanho (len) do mesmo, sendo enviada no fim uma mensagem com o nome do ficheiro e a indicar o seu tamanho em bytes. No fim é devolvido o file_to_chk pela função.

Se não existir (False) é criada a mensagem do erro “File does not existe” e colocada em msg. De seguida é importada a função **pack_err** para ser criado um pack ERR que irá conter o op, o nº de erro da situação da não existência do ficheiro e a mensagem específica criada (msg), que é colocado em err_send, sendo este devolvido pela função.

```
def file_verify(file_to_chk):
    "Faz a verificação da existência do ficheiro e diz o seu tamanho em bytes"

    if isinstance(file_to_chk, list):
        file_to_chk = str(file_to_chk).strip("[]'")
    if isinstance(file_to_chk, str):
        file_to_chk = file_to_chk.encode()
    if os.path.isfile(file_to_chk) == True:
        f1_ = open(file_to_chk, 'rb')
        f_ = f1_.read(8192)
        filesize_send = len(f_)
        print("file {} have {} bytes.".format(file_to_chk, filesize_send))
        return True, file_to_chk
    else:
        msg = "File {} does not exist.".format(file_to_chk)
        op = 5
        err = 1
        err_send = tftp.pack_err(op, err, msg)
        not_exist = False
        return not_exist, err_send
```

-» def **open_file**

Esta função abre um ficheiro para ser enviado em pacotes.

Primeiro é utilizada a função *compile* do módulo *re* para compilar um padrão de uma expressão regular para o objecto pat_all, depois é utilizada a função *search* do módulo *re* que procura por esse padrão (contido em pat_all) no ficheiro recebido (filename), sendo igualmente usado o método *group* que agrupa numa string os resultados onde foi verificado o padrão da expressão regular, sendo enviado para o objecto check.

Se check tiver conteúdo (o padrão da expressão regular), retira-se o caractere “0” do ficheiro (filename), sendo o mesmo lido em bytes, enviado para f_ e devolvido pela função; se não tiver conteúdo (o padrão da expressão regular), apenas é feita a leitura em bytes do ficheiro (filename), sendo enviado para f_ e devolvido pela função.

No fim implementámos algumas situações de excepções:

-» **FileNotFoundError**, onde é criado um pacote ERR com a função **pack_err**, sendo devolvido no fim;
-» **UnboundLocalError**;
-» **AttributeError**, onde é criado um pacote ERR com a função **pack_err**, sendo devolvido no fim.

```
def open_file(filename):
    "Abre um ficheiro para ser enviado em pacotes"
    try:
        pat_all = re.compile(b"([\x00-\x05])")
        if isinstance(filename, bytes) == False:
            filename = filename.encode()
        check = re.search(pat_all, filename).group()
        if check:
            file_O, file_w = filename.split(b'\x00')
            file = file_O
            f_ = open(file, 'rb').read()
            return f_
        else:
            file = filename
            f_ = open(file, 'rb').read()
            return f_

    except FileNotFoundError:
        print('File not found!')
        op = 5
        error = 1
        msg =b'File not found!'
        pac_err = pack_err(op, error, msg)
        return pac_err

    except UnboundLocalError:

        file = filename
        f_ = open(file, 'rb').read()
        return f_

    except AttributeError:
        if filename == None:
            print('File not found!')
            op = 5
            error = 1
            msg ='File not found'
            pac_err = pack_err(op, error, msg)
            return pac_err
        else:
            file = filename
            f_ = open(file, 'rb').read()
            return f_
```

-» def **treat_RQQ**

Esta função faz o tratamento / leitura do pacote RRQ.

Primeiro, faz a descompactação do pacote RRQ (data) e envia para o objecto lista. Depois é verificado o número de itens desse objecto, se tem 3 ou 4. Se tiver 3 itens, as variáveis op, file e mode receberão os respectivos dados, isto é, o opcode, o nome do ficheiro e o modo de transferência. Se tiver 4 itens, o quarto item a mais irá para a variável name_f, neste caso é o nome especificado do ficheiro no receptor, por parte do utilizador.

Seguidamente é usada a função **file_verify** para verificar a existência do ficheiro (file). Se existir, é utilizada a função **open_file** no ficheiro (file), seguindo-se a produção dos blocos, gerados pela função **chunks**. O método *next* vai devolver todos os valores do gerador, até que não haja mais nenhum. Individualmente é criado um pacote de DAT para cada um desses

blocos de dados. Se não existir o ficheiro (file) é produzido um pacote ERR com a mensagem da situação “File not found”, sendo devolvido pela função.

```
def treat_RQQ(data):
    "Tratamento / Leitura do pack RRQ"
    lista = tftp.unpack_R_W(data)
    if len(lista) == 3:
        op, file, mode = lista
        file = file
    if len(lista) == 4:
        op, file, name_f, mode = lista
        file1 = file, name_f
    IF_file, file = tftp.file_verify(file)
    if IF_file == True:
        file_O = tftp.open_file(file)
        print("Preparing '%s'." % (file))
        blocks = tftp.chunks(file_O, 512)
        info_file = next(blocks,'end')
        if info_file == 'end':
            print("END OF DAT ", file)
        info_file += b'\0'
        numb_blk = next(blocks, 'end')
        packet_DAT = tftp.pack_3_(op, numb_blk, info_file)
        return  file_O, file
    else:
        err = 1
        msg = b'File not found!'
        send_err = tftp.pack_err(op, err, msg)
        return send_err, msg, err
```

-» def **treat_WRQ**

Esta função faz o tratamento / leitura do pacote WRQ.

Primeiro, faz a descompactação do pacote WRQ (data) e envia para o objecto lista. Depois é verificado o número de itens desse objecto, se tem 3 ou 4. Se tiver 3 itens, as variáveis op, file e mode receberão os respectivos dados, isto é, o opcode, o nome do ficheiro e o modo de transferência.

Seguidamente é usada a função **file_verify** para verificar a existência do ficheiro (file). Se existir, é produzido um pacote ERR com a função **pack_err**, com a mensagem da situação “File already exists!”, sendo devolvido pela função. Se não existir, é produzido um pacote DAT com a função **pack_4_**, sendo devolvido pela função.

Se tiver 4 itens, o quarto item a mais irá para a variável name_f, neste caso é o nome especificado do ficheiro no receptor, por parte do utilizador. De seguida, é usada a função **file_verify** para verificar a existência do ficheiro (file). Se existir, é produzido um pacote ERR com a função **pack_err**, com a mensagem da situação “File already exists!”, sendo devolvido pela função. Se não existir, é produzido um pacote DAT com a função **pack_4_**, sendo devolvido pela função.

```
def treat_WRQ(data):
    "Tratamento / Leitura do pack WRQ"
    lista = tftp.unpack_R_W(data)
    if len(lista) == 3:
        op, file, mode = lista
        file = lista[1]
        IF_file, namefile = tftp.file_verify(file)
        if IF_file == True:
            op = 5
            err = 1
            msg = b'File already exists!'
            send_err = tftp.pack_err(op, err, msg)
            return send_err, err
        else:
            op = 4
            blk = 0
            ack_toSend = tftp.pack_4_(op, blk)
            return ack_toSend, file

    if len(lista) == 4:
        op, file, name_f, mode = lista
        file = lista[1]
        name_f = lista[2]
        IF_file, namefile = tftp.file_verify(file)
        if IF_file == True:
            op = 5
            err=1
            msg = b'File already exists!'
            send_err = tftp.pack_err(op, err, msg)
            return send_err, msg, err
        else:
            op = 4
            blk = 0
            ack_toSend = tftp.pack_4_(op, blk)
            return ack_toSend, file, name_f
```

-» def **treat_DAT1**

Esta função trata do primeiro bloco de dados recebido (DAT nº1).

É utilizada a função **unpack_data** para descompactar os dados para o objecto dat. De seguida, as variáveis op, blk e dat_r recebem os dados respectivos, isto é, o opcode, o nº do bloco do pacote DAT e os primeiros dados do ficheiro. Esses dados vão sendo escritos para o ficheiro filename_tosave.

Depois é produzido o pacote ACK com a função **pack_4_** e é devolvido no fim pela função.

```
def treat_DAT1(data, path, filename_tosave = b'filesaved_.txt'):
    "Tratamento / Leitura do pack DAT nº1 após RRQ"
    dat = tftp.unpack_dat(data)
    op, blk, dat_r = dat
    blk_num = blk
    file_write = open(filename_tosave, 'wb').write(dat_r)
    print('Bloco == a 1 -->', filename_tosave)
    print("File created --> %s" % (filename_tosave))
    op = 4
    send_ack = tftp.pack_4_(op, blk)
    print("PACKET ACK: ", send_ack)
    return send_ack, filename_tosave
```

-» def **treat_DAT2**

Esta função trata do(s) seguinte(s) bloco(s) de dados recebido (DAT n > 1).

É utilizada a função **unpack_data** para descompactar os dados para o objecto dat. De seguida, as variáveis op, blk e dat_r recebem os dados respectivos, isto é, o opcode, o nº do bloco do pacote DAT e os próximos dados do ficheiro. Esses dados vão sendo escritos para o ficheiro filename_tosave.

Depois é produzido o pacote ACK com a função **pack_4_** e é devolvido no fim pela função.

```
def treat_DAT2(data, path,filename_tosave):
    dat = tftp.unpack_dat(data)
    op, blk, dat_r = dat
    blk_num = blk
    print('num_block Treat_Dat > 1 --', blk_num)
    print('Bloco > a 1 -->', filename_tosave)
    file_write = open(filename_tosave, 'ab').write(dat_r)
    print("Written %s bytes on %s" % (file_write, filename_tosave))
    print("OP received --> ", op)
    print("BLK received --> ", blk)
    op = 4
    send_ack = tftp.pack_4_(op, blk)
    print("PACKET ACK: ", send_ack)
    return send_ack, filename_tosave
```

-» def **is_valid_ipv4_address**

Esta função analisa se o endereço IP é válido.

A função inet_pton do módulo socket converte o endereço (address) da família de endereços AF_INET para o formato binário, devolvendo True. Se não for um endereço desta família, será dada a indicação da excepção socket error, devolvendo False, logo não é um endereço válido.

```
def is_valid_ipv4_address(address):
    try:
        socket.inet_pton(socket.AF_INET, address)
    except AttributeError:  # No inet_pton here, sorry
        try:
            socket.inet_aton(address)
        except socket.error:
            return False
        return address.count('.') == 3
    except socket.error:  # Not a valid address
        return False

    return True
```

-» def **treat_ERR**

Esta função faz a leitura do pacote ERR e identifica qual é a mensagem de erro respectiva.

Os dados que vêm no pacote ERR (data) são colocados nas variáveis op e err, isto é, o opcode e o número do erro. A função depois analisa o número do erro em err, mostrando a mensagem respectiva para esse número.

```
def treat_ERR(data):
    "Tratamento / Leitura do pack ERR"
    print('ERR received...')

    ERR1 = b'\x00\x01'
    ERR2 = b'\x00\x02'
    ERR3 = b'\x00\x03'
    ERR4 = b'\x00\x04'
    ERR5 = b'\x00\x05'
    ERR6 = b'\x00\x06'
    ERR7 = b'\x00\x07'
    ERR8 = b'\x00\x08'

    op, err = data

    if err == ERR1:
        msg = 'File not found'
        return msg
    if err == ERR2:
         msg = 'Access violation'
         return msg
    if err == ERR3:
        msg = 'Disk full or allocation exceeded'
        return msg
    if err == ERR4:
        msg = 'Ilegal TFTP operation'
        return msg
    if err == ERR5:
        msg = 'Unknown transfer ID'
        return msg
    if err == ERR6:
        msg = 'File already exists'
        return msg
    if err == ERR7:
        msg = 'No such user'
        return msg
    if err == ERR8:
        msg = 'Terminate transfer due to option negotiation'
        return msg
```

As últimas funções são em relação ao DIR.

-» def **dir_pack_send**

Esta função cria um pacote DIR para ser enviado do cliente para o servidor, quando este escreve o comando DIR na prompt (é solicitado um ficheiro sem nome).

O pacote DIR (packet_fmt) é criado com a função *pack* do módulo *struct*, tendo em conta o formato definido em fmt, sendo devolvido no fim pela função.

```
def dir_pack_send(filename = b''):
    "Constrói um pacote DIR"
    op = 6
    num_blk = 1
    fmt = '!HH{}s'.format(len(filename))
    packet_fmt = struct.pack(fmt, op, num_blk, filename)
    return packet_fmt
```

-» def **dir_unpack**

Esta função descompacta o pacote DIR.

O pacote DIR (pack) é descompactado com a função *unpack* do módulo *struct*, tendo em conta o formato definido em fmt, sendo enviada a informação respectiva para as variáveis op (o opcode), num_blk (número do bloco) e dir_msg (a listagem do server). De seguida é feito o *strip* do /n (mudança de linha) ao *dir_msg* e enviado para o dir_msg2. Esta última variável, juntamente com a op e num_blk são colocadas num tuple com o nome lista, sendo este último devolvido pela função no fim.

```
def dir_unpack(pack):
    "Descompacta pack do tipo DIR"
    op = pack[:2]
    num_blk = pack[2:4]
    dir_msg = pack[4:]
    fmt = '!HH{}s'.format(len(dir_msg))
    op, num_blk, dir_msg = struct.unpack(fmt, pack)
    dir_msg = dir_msg.decode().strip('\x00')
    dir_msg2 = dir_msg.split('\n')
    lista = op, num_blk, dir_msg2
    return lista
```

-» def **make_dir**

Esta função faz a listagem do servidor.

```
def make_dir(path):
    lista_dir = os.popen('ls -lah %s' % (path)).read()
    return lista_dir
```

-» def **show_dir**

Esta função mostra a listagem do servidor.

```
def show_dir(dir_show):
    "Faz o DIR após a sua recepção"
    for i in dir_show:
        print(i)
```

Client.py

Importámos os módulos *socket*, *sys*, *struct*, *re*, *os.path*, *docopt,* e o *tftp* (ficheiro que contem as funções desenvolvidas para este projecto).

Importámos igualmente do módulo cmd para desenvolver uma shell que permite ao utilizador trabalhar com o programa cliente de forma interactiva.

Utilizámos a função *socket* do módulo *socket* para criar o mecanismo de socket para receber a ligação, passando dois argumentos: AF_INET (a família do protocolo) e SOCK_DGRAM (o tipo do socket, neste caso UDP).

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Utilizámos o *docopt neste projecto* para fazer a leitura da linha de comandos.

De seguida é verificada a validade do endereço IP introduzido na linha de comandos, que vai para o argumento <server>, utilizando a função **is_valid_ipv4_address** do ficheiro TFTP; no caso de não ser válido aparecerá a mensagem a indicar esse facto.

```
host_arg = args['<server>']
ip_verify = args['<server>'].split('.')
if len(ip_verify) == 1:
    pass
if len(ip_verify) == 4:
    verify_ip = tftp.is_valid_ipv4_address(args['<server>'])
    if verify_ip == False:
        print('The inserted server dont have a valid IP address.')
        sys.exit()
if 4 > len(ip_verify) > 1:
    print('The inserted server dont have a valid IP address.')
    sys.exit()
```

Se tiver sido colocado o argumento ‘get’ na linha de comandos, a variável filename recebe o <source_file> introduzido pelo utilizador, sendo a mesma codificada com o *encode* e no fim é acrescentado o caracter “0” em bytes no fim. Se for definido um novo nome para o ficheiro original, ou seja, se existir o argumento <dest_file>, este será igualmente codificado com o *encode* e adicionado ao filename.

No fim é importada a função **pack_R_W** para construir um pacote RRQ, utilizando as variáveis op (= 1), filename e mode (= octet) como parâmetros.

```
if args['get']:
    op = 1
    filename = args['<source_file>']
    filename = filename.encode()
    filename += b'\0'
    if args['<dest_file>']:
        filename += naming_file.encode()
    packet = tftp.pack_R_W(op, filename, mode)
```

Se tiver sido colocado o argumento 'put' na linha de comandos, a variável filename recebe o <source_file> introduzido pelo utilizador. Seguidamente é importada a função **file_verify** para verificar se existe ou não um ficheiro com esse nome no cliente. Se não existir esse ficheiro é dada uma mensagem a informar essa situação e é fechado o Python. Se existir o ficheiro, a variável filena recebe o nome do mesmo, sendo adicionado o caractere "0" no fim em bytes. Se for definido um novo nome para o ficheiro original, ou seja, se existir o argumento <dest_file>, este será codificado com o *encode* e adicionado ao filena.

No fim é importada a função **pack_R_W** para construir um pacote WRQ, utilizando as variáveis op (= 2), filena e mode (= octet) como parâmetros.

```
if args['put']:
    op = 2
    filename = args['<source_file>']
    print('filename', filename)
    file, namefile = tftp.file_verify(filename)
    if file == True:
        filena = namefile
        filena += b'\0'
    if file != True:
        print('The file %s does not exist' % args['<source_file>'])
        sys.exit()
    if args['<dest_file>']:
        filena += naming_file.encode()
    packet = tftp.pack_R_W(op, filena, mode)
```

A seguir desenvolvemos uma classe chamada MyPrompt com o atributo Cmd, que contem as funções para os comandos **get**, **put**, **dir**, **help** e **quit**, usados no programa no modo interactivo.

```
class MyPrompt(Cmd):
```

Função **do_get**:

Inicialmente é verificado o número de argumentos após o 'get', se for **0** o utilizador recebe a mensagem de *usage* a indicar a sintaxe correcta; se for **1** a variável filename recebe esse argumento, ou seja o nome do ficheiro, sendo o mesmo codificado com o *encode* e adicionado o caracter "0" em bytes no fim; se for **2**, quer dizer que o utilizador definiu um novo nome para o ficheiro original, sendo esse nome igualmente codificado e é adicionado ao filename.

No fim é importada a função **pack_R_W** para construir um pacote RRQ, utilizando as variáveis op (= 1), filename e mode (= octet) como parâmetros, sendo este enviado para o servidor, ao mesmo tempo o utilizador recebe uma mensagem do envio.

```
def do_get(self, args: list = sys.argv, addr = addr):
        args2 = args.split()
        if len(args2) == 0:
            print('usage: get ficheiro_remoto [ficheiro_local]')
        if len(args2) == 1:
            filename = args2[0].encode()
            filename += b'\0'
        if len(args2) == 2:
            filename = args2[0].encode()
            filename += b'\0'
            filename += args2[1].encode()
            name_to_save = args2[1]
        op = 1
        file = tftp.pack_R_W(op, filename, mode)
        s.sendto(file, addr)
        if s:
            print("sending RRQ...")
```

Depois foi feito um ciclo while para o cliente poder tratar de forma correcta o(s) pacote(s) DAT que irá ou não receber do servidor, após o pedido de RRQ.

Se recebeu um pacote do servidor, é verificado que tipo de pacote é, para isto é utilizada a função **check_pack**. Se for um pacote DAT, neste caso o DAT nº 1, é utilizada a função **treat_DAT1** onde é produzido um pacote ACK, permitindo ao cliente enviá-lo para o servidor para o informar da recepção do DAT nº 1. Se o nº do bloco do DAT for > 1, é utilizada a função **treat_DAT2** onde é produzido um pacote ACK, permitindo ao cliente enviá-lo para o servidor para o informar da recepção do DAT nº > 1.

```
try:
        while 1:
            data, addr = s.recvfrom(16384)
            s.settimeout(10.0)
            pack_type = tftp.check_pack(data)
            print("PACKET TYPE :", pack_type)
            if data == 'end':
                s.settimeout(10)

            if pack_type == DAT:
                blk = data[2:4]
                if len(args2) == 2:
                    if blk == BLK1:
                        ack_send, namesaved = tftp.treat_DAT1(data, path, name_to_save)
                        print('name --> ', namesaved)
                        s.sendto(ack_send, addr)
                    if blk > BLK1:
                        ack_send, namesaved = tftp.treat_DAT2(data, path, namesaved)
                        print('name2 --> ', namesaved)
                        s.sendto(ack_send, addr)
                else:
                    if blk == BLK1:
                        ack_, namesaved = tftp.treat_DAT1(data, path)
                        s.sendto(ack_, addr)
                    if blk > BLK1:
                        ack_send, namesaved = tftp.treat_DAT2(data, path, namesaved)
                        s.sendto(ack_, addr)
```

No caso de receber um pacote ERR (uma situação de erro), é utilizada a função **unpack_err** para descompactar esse pacote (data), recebendo no fim o nº e a respectiva mensagem do erro.

```
# LEITURA DO PACOTE 5 (ERR)
if pack_type == ERR:
                info = tftp.unpack_err(data)
                op, err, msg = info
                print('Error %d! %s' % (err, msg))
                continue
```

Foi igualmente inserida a excepção **socket.timeout**.

```
except socket.timeout:
            print('Turning  off connection...')
            s.connect(addr)
            s.settimeout(7)
            prompt = MyPrompt()
            prompt.prompt = 'tftp client> '
            prompt.cmdloop()
```

Função **do_put**:

Inicialmente é verificado o número de argumentos após o 'put', se for **0** o utilizador recebe a mensagem de *usage* a indicar a sintaxe correcta; se for **1** a variável filename recebe esse argumento, ou seja o nome do ficheiro, sendo o mesmo codificado com o *encode* e adicionado o caracter "0" em bytes no fim; se for **2**, quer dizer que o utilizador definiu um novo nome para o ficheiro original, sendo esse nome igualmente codificado e é adicionado ao filename.

Seguidamente é importada a função **file_verify** para verificar se existe ou não um ficheiro com esse nome no cliente. Se não existir esse ficheiro é dada uma mensagem a informar essa situação; se existir esse ficheiro, é importada a função **pack_R_W** para construir um pacote WRQ, utilizando as variáveis op (= 2), filename e mode (= octet) como parâmetros, sendo este enviado para o servidor.

```
def do_put(self, args: list = sys.argv, addr = addr):

        args2 = args.split()

        if len(args2) == 0:
            print('usage: put ficheiro_local [ficheiro_remoto]')
            prompt = MyPrompt()
            prompt.prompt = 'tftp client> '
            prompt.cmdloop()

        if len(args2) == 1:
            filename = args2[0].encode()
            filename += b'\0'
            file_to_verify = args2[0]
            print('filename', filename)

        if len(args2) == 2:
            filename = args2[0].encode()
            filename += b'\0'
            filename += args2[1].encode()
            file_to_verify = args2[0]
        ct4 = 0
        op = 2
        file, namefile = tftp.file_verify(file_to_verify)

        if file == True:
            packet = tftp.pack_R_W(op, filename, mode)
            s.sendto(packet, addr)

        if file != True:
            print('The file %s does not exist' % (file_to_verify))
            prompt = MyPrompt(host, port)
            prompt.prompt = 'tftp client> '
            prompt.cmdloop()
```

Depois foi feito um ciclo while para o cliente poder tratar de forma correcta o(s) pacote(s) ACK que irá ou não receber do servidor, após o pedido de WRQ e após o envio do(s) pacote(s) DAT.

Se recebeu um pacote do servidor, é verificado que tipo de pacote é, para isto é utilizada a função **check_pack**. Se o pacote for um ACK, é importada a função **open_file** para abrir o namefile que contem o nome do ficheiro que se quer enviar do cliente para o servidor, de seguida é utilizada a função **chunks** para gerar o(s) bloco(s) de dados desse ficheiro para serem colocados num pacote DAT que se vai produzir com a função **pack_3_**, sendo no fim enviado esse pacote DAT para o servidor.

Este ciclo decorre enquanto houver blocos dados do ficheiro para enviar para o servidor, enquanto continuar a receber os pacotes ACK do servidor e também se não ocorrer nenhum erro. Quando o ficheiro tiver sido enviado na sua totalidade, será dada a informação do envio do ficheiro.

```
try:
        while 1:
            data, addr = s.recvfrom(16384)
            s.settimeout(10.0)
            pack_type = tftp.check_pack(data)
            print("PACKET TYPE :", pack_type)
            if data == 'end':
                s.settimeout(10)

            if pack_type == ACK:
                op = 3
                ct4 += 1
                if ct4 == 1:
                    fi = tftp.open_file(namefile)
                    blocks = tftp.chunks(fi, 512)
                inf = next(blocks, 'end')
                file = str(file).strip("[]")
                if inf == 'end':
                    print("File  %s has been sent." % (namefile))
                    ct4 = 0
                    s.settimeout(10)
                    continue
                numb_blk = next(blocks, 'end')
                packet = tftp.pack_3_(op, numb_blk, inf)
                s.sendto(packet, addr)
```

No caso de receber um pacote ERR (uma situação de erro), é utilizada a função **unpack_err** para descompactar esse pacote (data), recebendo no fim o nº e a respectiva mensagem do erro.

```
# LEITURA DO PACOTE 5 (ERR)
if pack_type == ERR:
                info = tftp.unpack_err(data)
                op, err, msg = info
                print('Error %d! %s' % (err, msg))
                continue
```

Foi igualmente inserida a excepção **socket.timeout**.

```
except socket.timeout:
        print('Turning  off connection...')
        s.connect(addr)
        s.settimeout(10)
        prompt = MyPrompt()
        prompt.prompt = 'tftp client> '
        prompt.cmdloop('Starting prompt...')
```

Função **do_dir**:

É importada a função **dir_pack_send** para criar um pacote DIR para ser enviado do cliente para o servidor.

Se receber um pacote do servidor em resposta, é utilizada a função **check_pack** para verificar se o pacote é um DIR. Se for um DIR é importada a função **dir_unpack** para fazer o descompactar do pacote, seguidamente é utilizada a função **show_dir** para mostrar a listagem do servidor. Está definido um timeout de 60 segundos de espera, pelo retomar da comunicação entre o cliente e o servidor.

```
def do_dir(self, args = sys.argv):
        filename = b''
        dir_send = tftp.dir_pack_send(filename)
        s.sendto(packet, addr)

        try:
            while 1:

                data, addr = s.recvfrom(16384)
                pack_type = tftp.check_pack(data)
                op = data[:2]
                blokc = data[2:4]

                if op == DIR:
                    op, numb, dir_msg = tftp.dir_unpack(data)
                    tftp.show_dir(dir_msg)

                if not data:
                    s.settimeout(60)
```

Foram inseridas duas excepções:

-» **socket.timeout**, com um determinado timeout;
-» **KeyboardInterrupt**, para o caso do utilizador utilizar por exemplo a combinação “CTRL + C”, que levará a que seja terminada a ligação ao servidor.

```
except socket.timeout:
            print('Turning  off connection...')
            s.connect(addr)
            s.settimeout(5)
            prompt = MyPrompt()
            prompt.prompt = 'tftp client> '
            prompt.cmdloop('Starting prompt...')

except KeyboardInterrupt:
            print ("Bye")
            sys.exit()
```

Função **do_help**:

```
def do_help(self, args):
        "Show commands"
        print("No help available yet")
```

Função **do_quit**:

Para terminar a ligação é feito o *raise* do SystemExit.

```
def do_quit(self, args):
        print("Quitting tftp.")
        raise SystemExit
```

Depois foi definida a inicialização da prompt com [-p serv_port] <server>, quando os argumentos **put**, **get**, **<source_file>** e **<dest_file>** são False ou None.

```
if args['put'] == False:
    ct_args += 1

if args['get'] == False:
    ct_args += 1

if args['<source_file>'] == None:
    ct_args += 1

if args['<dest_file>'] == None:
    ct_args += 1

if ct_args == 4:
    prompt = MyPrompt(host, port)
    prompt.prompt = 'tftp client> '
    prompt.cmdloop('Starting prompt...')
```

Nota: Termina aqui o código referente ao funcionamento do cliente em modo interactivo.

Para finalizar, foi criado um ciclo while para definir a forma como o cliente procede, no modo não-interactivo, consoante o pacote que recebe. É importada a função **check_pack** para verificar o tipo de pacote.

```
while True:
        try:
            data, addr = s.recvfrom(buf)
            pack_type = tftp.check_pack(data)
            op = data[:2]
            bloco = data[2:4]

            if bloco != ERR:
                s.settimeout(60)
```

Se for um pacote DAT, neste caso o DAT nº 1, é utilizada a função **treat_DAT1** onde é produzido um pacote ACK, permitindo ao cliente enviá-lo para o servidor para o informar da recepção do DAT nº 1. Se o nº do bloco do DAT for > 1, é utilizada a função **treat_DAT2** onde é produzido um pacote ACK, permitindo ao cliente enviá-lo para o servidor para o informar da recepção do DAT nº > 1.

```
# LEITURA DO PACOTE 3 (DAT)
if pack_type == DAT:
            blk = data[2:4]
            if args['<dest_file>']:
                if blk == BLK1:
                    ack_send, namesaved = tftp.treat_DAT1(data, path, args['<dest_file>'])

                    s.sendto(ack_send, addr)
                if blk > BLK1:
                    ack_send, namesaved = tftp.treat_DAT2(data, path, namesaved)

                    s.sendto(ack_send, addr)
            else:
                if blk == BLK1:
                    ack_, namesaved = tftp.treat_DAT1(data, path, args['<source_file>'])
                    s.sendto(ack_, addr)
                if blk > BLK1:
                    ack_send, namesaved = tftp.treat_DAT2(data, path, namesaved)
                    s.sendto(ack_, addr)
```

Se for um pacote ACK, consoante o nº do pacote ACK, o cliente actua de determinada forma.

Se o pacote ACK for o nº 0, ou seja, o pacote de resposta do servidor, após um pedido de WRQ por parte do cliente ao servidor, é utilizada a função **open_file** para abrir o filename que contem o nome do ficheiro que se quer enviar do cliente para o servidor, de seguida é utilizada a função **chunks** para gerar o primeiro bloco de dados desse ficheiro para ser colocado num pacote DAT que se vai produzir com a função **pack_3_**, sendo no fim enviado esse pacote DAT para o servidor.

Se o pacote ACK for o nº 1, ou seja, o pacote de resposta do servidor, após a recepção do primeiro pacote DAT enviado pelo cliente, é novamente utilizada a função **open_file** para abrir o filename que contem o nome do ficheiro que se quer enviar do cliente para o servidor, de seguida é utilizada a função **chunks** para gerar o próximo bloco de dados desse ficheiro para ser colocado no próximo pacote DAT que se vai produzir com a função **pack_3_**, sendo no fim enviado esse pacote DAT para o servidor.

Se o pacote ACK for > 1, continuarão a ser gerados os blocos de dados do ficheiro a enviar, com a função **chunks**, enquanto existirem dados para enviar, para se completar a transferência. No fim é sempre produzido o respectivo pacote DAT com a função **pack_3_** e enviado ao servidor.

Este ciclo decorre enquanto houver blocos dados do ficheiro para enviar para o servidor, enquanto continuar a receber os pacotes ACK do servidor e também se não ocorrer nenhum erro.

```
# LEITURA DO PACOTE 4 (ACK)
if pack_type == ACK:
            op = data[:2]
            blk = data[2:]

            if blk <= BLK1:

                if blk == BLK0:
                    ct4 += 1
                    file_open = tftp.open_file(filename)

                    gen = tftp.chunks(file_open, 512)
                    dat = next(gen, 'end')

                    if dat == 'end':
                        print("END OF DAT ", filename)
                        ct4 = 0
                        continue
                    op = 3
                    blocks = next(gen, 'end')
                    packet_DAT = tftp.pack_3_(op, blocks, dat)
                    s.sendto(packet_DAT, addr)
                    continue

                if blk == BLK1:
                    if ct4 == 0:
                        file_open = tftp.open_file(filename)
                        gen = tftp.chunks(file_open, 512)
                        dat = next(gen, 'end')
                        print('DAT', dat)
                        if dat == 'end':
                            print("File %s sended with %d " % (filename, len(file_open)))
                            ct4 = 0
                            continue
                        op = 3
                        blocks = next(gen, 'end')
                        packet_DAT = tftp.pack_3_(op, blocks, dat)
                        s.sendto(packet_DAT, addr)
                        continue

            if blk > BLK1:

                dat = next(gen, 'end')
                if dat == 'end':
                    print("File %s sended with %d " % (filename, len(file_open)))
                    ct4 = 0
                    continue
                blocks = next(gen, 'end')
                op = 3
                packet_DAT = tftp.pack_3_(op, blocks, dat)
                s.sendto(packet_DAT, addr)
                continue
```

No caso de receber um pacote ERR (uma situação de erro), é utilizada a função **unpack_err** para descompactar esse pacote (data), recebendo no fim o nº e a respectiva mensagem do erro.

```
# LEITURA DO PACOTE 5 (ERR)
if pack_type == ERR:
        info = tftp.unpack_err(data)
        op, err, msg = info
        print('Error %d! %s' % (err, msg))
        s.settimeout(5)
        continue
```

Foram inseridas quatro excepções:

-» **ConnectionRefusedError**;
-» **socket.timeout**, com um determinado timeout;
-» **TypeError**;
-» **KeyboardInterrupt**, para o caso do utilizador utilizar por exemplo a combinação “CTRL + C”, que levará a que seja terminada a ligação ao servidor.

```
except ConnectionRefusedError:

        print("Could not open the socket for the host %s with IP address '%s'." % (host_arg, host))
        print("Setting timeout. Trying...")
        s.connect(addr)
        ct = 0
        s.connect(addr)
        s.settimeout(10)

except socket.timeout:

        if blk == 0:
            print("Turning off the connection with '%s'." % (host_arg))
            break
        if blk > BLK1:
            print("Turning off the connection with the host address '%s'." % (host_arg))
            s.settimeout(10)
            print('Bye...')
            break
        else:
            print('Trying again...')
            ct += 1
            s.connect(addr)
            s.settimeout(10)
            if ct == 2:
                print("Can't connect!")
                break

except TypeError:
        print("Turning off the connection (typeerror) with the IP address '%s'." % (host_arg))

except KeyboardInterrupt:
        print ("Bye")
        break
```

Server.py

Importámos o módulo *socket*, a família de endereços *AF_INET*, o tipo de socket *SOCK_DGRAM*, a opção *SOL_SOCKET* e a *flag SO_REUSEADDR*.

Do módulo *socketserver* importámos as classes *BaseRequestHandler* e *ThreadingUDPServer*.

Importámos igualmente os módulos *sys*, *struct*, *re*, *os*, *os.path*, *docopt* e o *tftp* (ficheiro que contem as funções desenvolvidas para este projecto).

Utilizámos o *docopt neste projecto* para fazer a leitura da linha de comandos.

O *host* ficou definido com plicas vazias, ou seja para todas as interfaces.

Utilizámos a função *socket* do módulo *socket* para criar o mecanismo de socket para receber a ligação, passando dois argumentos: AF_INET (a família do protocolo) e SOCK_DGRAM (o tipo do socket, neste caso UDP).

```
host = ''  # Symbolic name meaning all available interfaces
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Definimos o *SOL_SOCKET* como opção do socket e o *SO_REUSEADDR* como flag para evitar o erro: **OSError: [Errno 98] Address already in use**.

Utilizámos a classe *ThreadingUDPServer* para permitir o reuso do endereço IP.

```
s.setsockopt(SOL_SOCKET, SO_REUSEADDR, True)
ThreadingUDPServer.allow_reuse_address = True
```

Depois tendo em conta o endereço (addr), o bind tentará estabelecer ligação do socket a esse mesmo endereço; consoante o resultado dessa tentativa, será dada uma mensagem específica.

```
try:
    s.bind(addr)
    print('Socket bind complete')
except socket.error as msg:
    print('Unable to bind to port %d' % (port))
    sys.exit()
```

De seguida foi criado um ciclo while, onde o servidor fica à escuta de pedidos por parte do cliente.

Aqui é importada a função *check_pack* para avaliar o pacote recebido pelo servidor, sendo devolvido o tipo do pacote, ou seja, o tipo do pedido.

As variáveis data, addr, host, port, op e bloco recebem os respectivos dados.

```
while True:
    try:
        print( "Waiting for requests on '%s' port '%s'\n" % (hostname, args['<port>']))
        data, addr = s.recvfrom(buf)
        host = addr[0]
        port = addr[1]
        print('port', port)
        print ('received %s bytes from %s' % (len(data), addr))
        pack_type = tftp.check_pack(data)
        print("RESULT :", pack_type)
        op = data[:2]
        bloco = data[2:4]
```

Se o pacote for um RRQ, é importada a função **treat_RQQ** para "tratar" desse pacote, sendo o resultado colocado na variável unpacked.

O servidor como resposta ao RRQ, cria um pacote DAT com a função **pack_3_** enviando o mesmo para o cliente.

```
# LEITURA DO PACOTE 1 (RRQ)
if pack_type == RRQ:
            unpacked = tftp.treat_RQQ(data)
            if len(unpacked) == 2:
                file_O, file = unpacked
                op = 3
                blocks = tftp.chunks(file_O, 512)
                info_file = next(blocks,'end')
                if info_file == 'end':
                    print("END OF DAT ", file)
                info_file += b'\0'
                numb_blk = next(blocks, 'end')
                packet_DAT = tftp.pack_3_(op, numb_blk, info_file)
                filen = file.decode()
                s.sendto(packet_DAT, addr)

            if len(unpacked) == 3:
                send_err, msg, err = unpacked
                s.sendto(send_err, addr)
```

Se o pacote for um WRQ, é importada a função **treat_WRQ** para "tratar" desse pacote, sendo o resultado colocado na variável output. Essa função durante a sua execução cria um pacote ACK para ser enviado pelo servidor para o cliente, como resposta ao pedido de WRQ.

```
# LEITURA DO PACOTE 2 (WRQ)
if pack_type == WRQ:
            output = tftp.treat_WRQ(data)
            if len(output) == 2:
                ack_send, filen = output
                if filen == False:
                    print('Error 1!')
            if len(output) == 3:
                ack_send, filen, file_to_save = output
                if filen == False:
                    print('Error %d! %s' % (file, file_to_save))

            s.sendto(ack_send, addr)
```

Se o pacote for um DAT e for o bloco nº 1, é importada a função **treat_DAT1**, se o nº do bloco for superior a 1, será importada a função **treat_DAT2**. Em qualquer uma das situações, as respectivas funções irão produzir um pacote ACK, permitindo ao servidor enviá-lo para o cliente para o informar da recepção do DAT.

```
# LEITURA DO PACOTE 3 (DAT)
if pack_type == DAT:
        blk = data[2:4]

        if len(output) == 2:
            if blk == BLK1:
                file_toW = filen[0]
                ack_send, namesaved =
tftp.treat_DAT1(data, args['<directory>'], file_toW)
            if blk > BLK1:
                ack_send, namesaved =
tftp.treat_DAT2(data, args['<directory>'], namesaved)
        if len(output) == 3:
            if blk == BLK1:
                ack_send, namesaved =
tftp.treat_DAT1(data, args['<directory>'], file_to_save)
            if blk > BLK1:
                ack_send, namesaved =
tftp.treat_DAT2(data, args['<directory>'], namesaved)
        s.sendto(ack_send, addr)
```

Se o pacote for um ACK, é importada a função **pack_3**_ onde é criado um pacote DAT do próximo bloco de dados do ficheiro, sendo enviado para o cliente.

```
# LEITURA DO PACOTE 4 (ACK)
if pack_type == ACK:
        op = 3
        ct4 += 1
        inf = next(blocks, 'end')
        file = str(file).strip("[]")
        if inf == 'end':
            print("END OF DAT! The file requested, %s has been sent.\n" % (filen))
            ct4 = 0
            continue
        numb_blk = next(blocks, 'end')
        packet = tftp.pack_3_(op, numb_blk, inf)
        s.sendto(packet, addr)
```

Se o pacote for um ERR, é importada a função **unpack_err**, onde o servidor receberá a informação do número e a mensagem do erro respectivo.

```
# LEITURA DO PACOTE 5 (ERR)
if pack_type == ERR:
        info = tftp.unpack_err(data)
        op, err, msg = info
        print('Error %d! %s' % (err, msg))
        continue
```

Se o pacote for um DIR, é importada a função **make_dir** onde é criada a listagem do conteúdo do servidor, sendo de seguida importada a função **dir_pack_send** para enviar essa listagem para o cliente.

```
# LEITURA DO PACOTE 6 (DIR)
if pack_type == DIR:
        path = args['<directory>']
        lista_dir = tftp.make_dir(path)
        send_dir = tftp.dir_pack_send(lista_dir)
```

No final do código estão definidas as excepções:

-» **ConnectionRefusedError**;
-» **socket.timeout**, onde está definido o temporizador de 10 segundos de timeout no socket;
-» **KeyboardInterrupt**, para o caso do utilizador utilizar por exemplo a combinação “CTRL + C”, que levará a que seja terminada a ligação ao servidor.

```
except ConnectionRefusedError:
    print("Error reaching the server %s with the ip %s." % (hostname, ip))

except socket.timeout:
    print('Trying again...')
    ct += 1
    s.connect(addr)
    s.settimeout(10)
    if ct == 2:
        print("Cant connect!")
        break

    break
except KeyboardInterrupt:
    print ("Bye")
    break

s.close()
```

Conclusão

Neste projecto, já conseguimos implementar o módulo docopt para fazer a leitura da linha de comandos, ao contrário do projecto anterior onde tivemos algumas dificuldades, acabando por optar pelo argparse.

Ao contrário do projecto anterior onde o mesmo foi desenvolvido em unibloco, neste conseguimos implementar def's e uma classe que também ela inclui def's para o modo interactivo.

Na construção dos programas, notou-se alguma melhoria na eficiência do desenvolvimento em tempo útil, uma vez que começámos a desenvolver este projecto, praticamente desde a altura que nos foi apresentado / explicado o que se pretendia para o mesmo. Acabámos por ficar mais vezes no Centro de Formação a fazer as ditas horas extra, que acabam sempre por ajudar, na troca de ideias entre os formandos.

Desenvolvemos o programa cliente, tal como o programa servidor, optando por criar um terceiro ficheiro onde inclui todas as def's desenvolvidas, tal como o Formador tinha sugerido, que acabou por revelar-se útil, principalmente para não tornar o código de cada um dos dois programas demasiado pesado visualmente, para nós que o estávamos a desenvolver e para quem o for ler a seguir.

Em relação à Segurança SSL / TLS acabámos só por fazer o anexo II sobre o assunto, não tendo sido implementado nos programas, por já estarmos muito em cima do prazo, mesmo tendo o mesmo sido alargado pelo Formador e também mediante os conhecimentos ainda um pouco aquém neste assunto, acabou por impedir avançar para a sua construção / implementação.

Dito isto, apesar dessa falta e do servidor apenas responder a um pedido de cada vez, os programas estão funcionais e com o comando DIR implementado.

No fim, sentimos que o nosso conhecimento no "mundo" Python foi reforçado com este projecto, que apesar de ter sido mais complicado / exigente que o anterior, não deixou de ser interessante, permitindo ao mesmo tempo perceber o funcionamento do protocolo TFTP.

Venha o próximo projecto.

Anexo I - UDP

O protocolo **UDP** (User Datagram Protocol) oferece (ao contrário do TCP) uma ligação não orientada à conexão, entre dois terminais, não oferecendo correção de erros, garantia de entrega, nem garantia de entrega pela ordem dos dados transmitidos. Desta forma, consegue ser mais rápido que o **TCP** (Transmission Control Protocol), pois só envia os dados, não tendo mais nenhuma preocupação. O UDP tem um *overhead* mínimo, sendo cada pacote (datagrama) constituído por um cabeçalho e por dados do utilizador.

Uma vez que não é orientado à conexão, podem ser enviados datagramas em qualquer altura sem que o destino tenha de ser avisado, não sendo igualmente necessário negociar uma ligação.

Os datagramas UDP (ao contrário do TCP), têm as suas fronteiras bem definidas, isto é, se um terminal enviar um datagrama com determinado conteúdo, será recebido no destino apenas um datagrama que contém esse conteúdo (isto se não se perder pelo caminho), não havendo ao nível da camada de transporte, segmentação de mensagens na origem ou fusão de datagramas no destino.

Este protocolo é portanto pouco fiável, apesar da taxa de falhas na Internet ser reduzida, aumentando nas redes sem fios, onde é mais fácil ocorrerem colisões e/ou interferências.

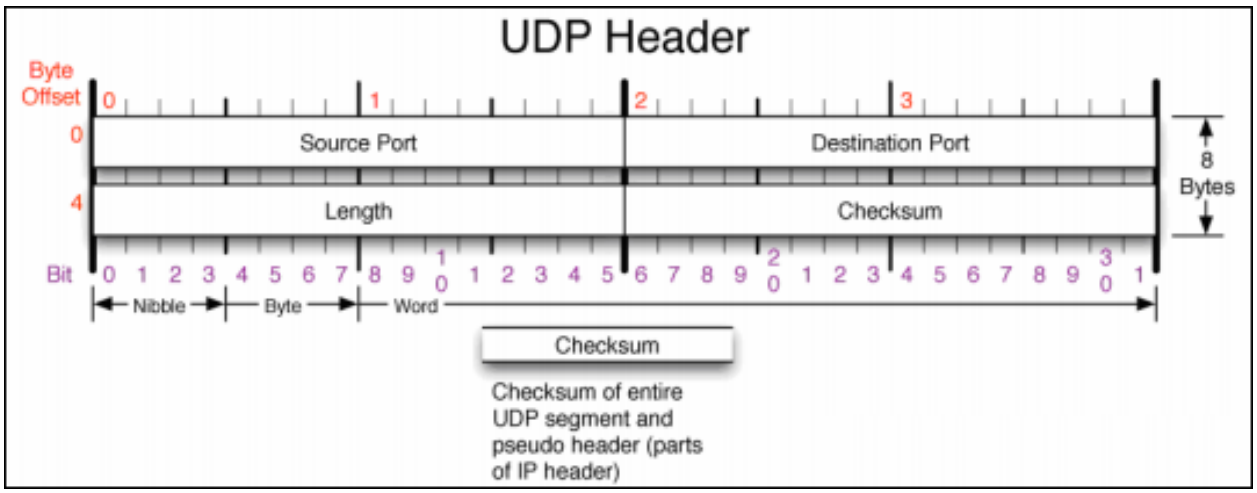


Imagem 1 - Formato do cabeçalho de uma trama UDP

Vantagens do UDP

-» O *overhead* do S.O. é menor do que o do TCP, assim como a latência no arranque da comunicação, uma vez que não existe estabelecimento de sessão nem o mecanismo de arranque lento associado ao TCP;

-» O receptor de um pacote UDP, recebe-o como foi enviado, com as fronteiras bem definidas;

-» Permite transmitir em *broadcast* e *multicast*;

-» Pode-se enviar/receber dados na mesma porta/socket de diferentes máquinas, uma vez que o endereço de destino é especificado por cada chamada ao sistema para enviar dados.

Desvantagens do UDP

-» Não existem garantias dos pacotes serem entregues, podendo os mesmos serem entregues várias vezes ou desordenadamente (o TCP garante que esses problemas não ocorram);

-» Não tem nenhum mecanismo de controlo de congestão, ao contrário do TCP;

-» Os datagramas UDP são os primeiros a serem descartados quando um computador está com falta de memória.

Observação geral sobre UDP vs TCP

Quando a ligação de rede é boa, o *overhead* introduzido pelo **TCP** é suficientemente pequeno para ser ignorado.

Em ligações fracas, o *overhead* do **TCP** é significativo, mas ao mesmo tempo a perda de pacotes do **UDP** torna-se um problema sério, implicando retransmissões para garantir-se que os dados chegaram ao destino.

Anexo II - Segurança com TLS/SSL

O protocolo **SSL** (Secure Sockets Layer) é um protocolo criptográfico, desenvolvido pela Netscape, que se baseia em cifras assimétricas (chave privada + chave pública) tendo como principal objectivo garantir a segurança e integridade dos dados que são transmitidos em redes inseguras (por ex. **Internet**), entre um servidor e um browser (cliente).

Quando um utilizador acede a um site que recorre ao SSL, o servidor envia ao cliente a chave pública (presente no certificado digital), que permite que o mesmo possa cifrar a informação que irá passar para o servidor. Assim que o servidor recebe essa informação, utiliza a sua chave privada (conhecida apenas pelo dono do certificado) para decifrar os dados transmitidos pelo cliente.

Existem várias aplicações para este protocolo, como por exemplo o comércio electrónico, servidores Web, servidores FTP, etc.

Um site seguro é facilmente identificado, uma vez que no seu URL aparece **https**:// em vez do normal **http**://, sendo que o https é um subprotocolo do protocolo http, mas com segurança no envio de informação.

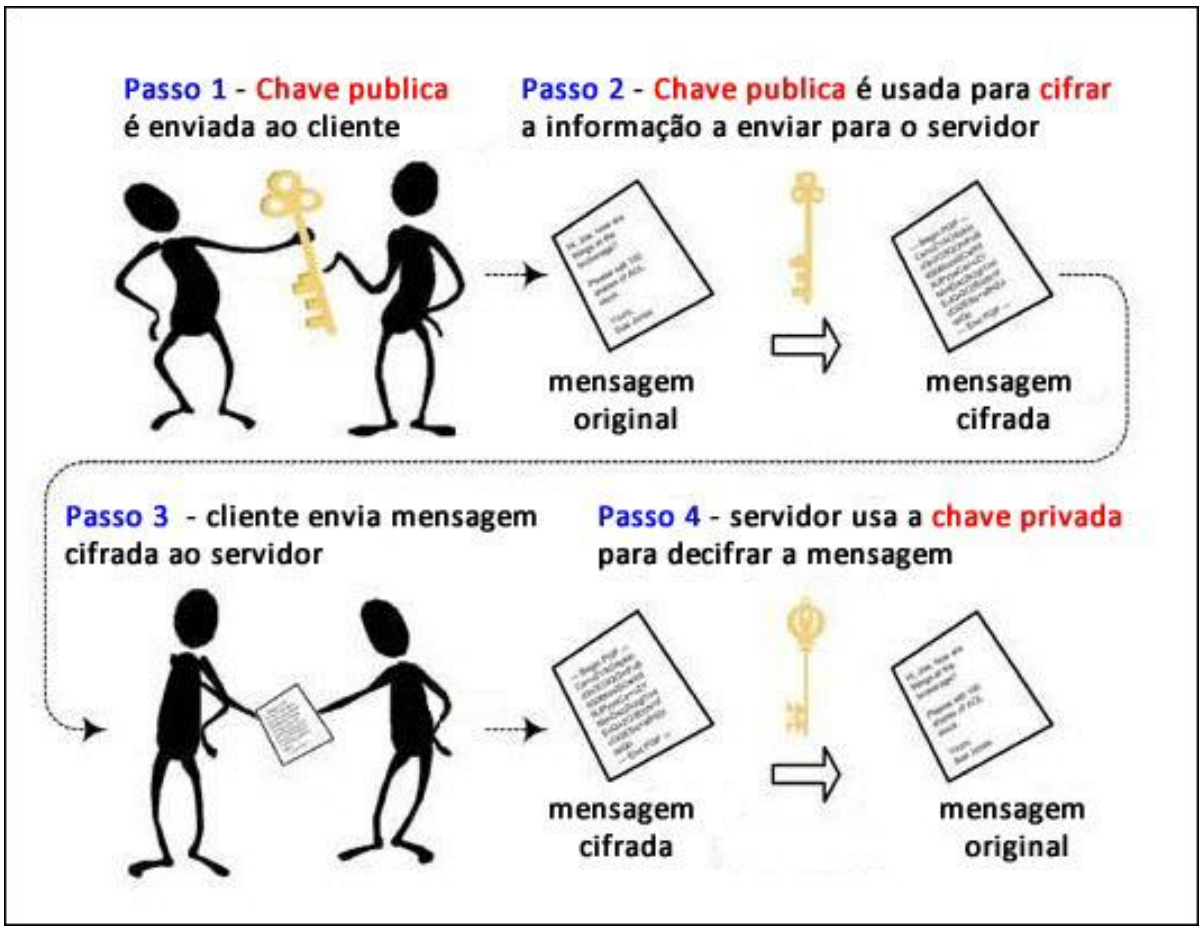


Imagem 2 – Chave pública e chave privada

Algoritmos criptográficos com SSL:

- **DES** (Data Encryption Standard) --» algoritmo de criptografia utilizado pelo governo americano;
- **DSA** (Digital Signature Algorithm) --» parte do padrão de autentificação digital usado pelo governo americano;
- **KEA** (Key Enchange Algorithm) --» algoritmo usado para a troca de chaves pelo governo americano;
- **MD5** (Message Digest Algorithm) --» algoritmo de criptografia digest;
- **RC2** / **RC4** (Rivest Encryption Ciphers) --» desenvolvido pela RSA Data Security;
- **RSA** --» Algoritmo de chave pública para criptografia e autentificação;
- **RSA Key Enchange** --» Algoritmo para troca de chaves usado em SSL baseado no algoritmo RSA (o mais comum);
- **SHA-1** (Secure Hash Algorithm) --» função hash usada pelo governo americano;
- **SKIPJACK** --» Algoritmo de chave simétrica implementado no hardware Fortezza;
- **Triple-DES** --» algoritmo DES aplicado três vezes.

O que é um certificado digital?

É um documento digital que contém informações de quem o possui, como o nome, morada, localidade, e-mail, duração do certificado, domínio e nome da entidade que assina o certificado. Contém igualmente uma chave pública e uma hash que possibilita verificar a integridade do próprio certificado.

A entidade certificadora (por ex. **Multicert**) confirma que o possuidor do certificado é quem afirma ser e assina o mesmo, impossibilitando a sua modificação.

O protocolo **TLS** (Transport Layer Security) é o sucessor do SSL, sendo utilizado mais frequentemente como uma configuração nos programas de e-mail, podendo tal como o SSL, ter um papel em qualquer transação cliente <--> servidor.

Bibliografia / Webgrafia

- http://docopt.org
- https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol
- https://pplware.sapo.pt/tutoriais/networking/redes-vamos-aprender-a-montar-um-servidor-tftp/
- https://docs.python.org/3/library/socket.html
- https://docs.python.org/3/library/re.html
- https://docs.python.org/3/library/struct.html