



PROJECTO 3 - TFTPy: CLIENTE TFTP (Beta)

INTRODUÇÃO E OBJECTIVOS

A finalidade deste projecto passa por desenvolver uma aplicação para transferência de ficheiros baseada no **Trivial File Transfer Protocol (TFTP)** e, de caminho, aprender a utilizar Python 3 e sockets para programar em redes.

Neste projecto pretende-se que desenvolva um cliente TFTP com interface da linha de comandos. Como extra, poderá também desenvolver um cliente com interface gráfica. Existem outros elementos do protocolo que são considerados mais avançados e cuja implementação é considerada um extra do projecto. Estes elementos, que são explicados ao longo do enunciado, são resumidos na secção **AValiação**. Nesta secção encontra também uma explicação sobre o que se entende por "extra" ou "opcional" no contexto deste projecto.

O PROTOCOLO

O TFTP é um protocolo para transferência de ficheiros, muito simples de implementar, que permite a um cliente receber ou enviar um ficheiro para um servidor remoto. Foi concebido para arrancar uma imagem remota de um sistema operativo em máquinas sem discos (*bootstrapping*). Para tal, utiliza o protocolo UDP da camada de transporte do modelo TCP/IP e o porto conhecido 69 (*well known port*). O TFTP não suporta os mecanismos de controlo de fluxo e detecção de erros utilizados em protocolos mais sofisticados. Em termos de funcionalidades, apenas suporta transferências de ficheiros entre cliente e servidor. Não permite transferência de directorias nem quaisquer outras operações para manipulação de ficheiros, como listagem, remoção, renomeação, etc. Também não fornece mecanismos de autenticação nem de confidencialidade, por isso deve ser utilizado apenas no contexto de uma rede local. Actualmente, o TFTP é o protocolo utilizado pelo mecanismo PXE (*Preboot Execution Environment*) e pelo protocolo BOOTP para carregar um sistema operativo a partir da LAN. Uma vez que o PXE integra a especificação UEFI 2.0, os novos *firmwares* baseados em UEFI incorporam um cliente de TFTP.

Pode encontrar uma descrição pormenorizada do protocolo em [1] e [2]. Neste documento listamos alguns dos aspectos e requisitos mais importantes da solução a implementar:

- O TFTP define 5 tipos de pacotes ou mensagens:

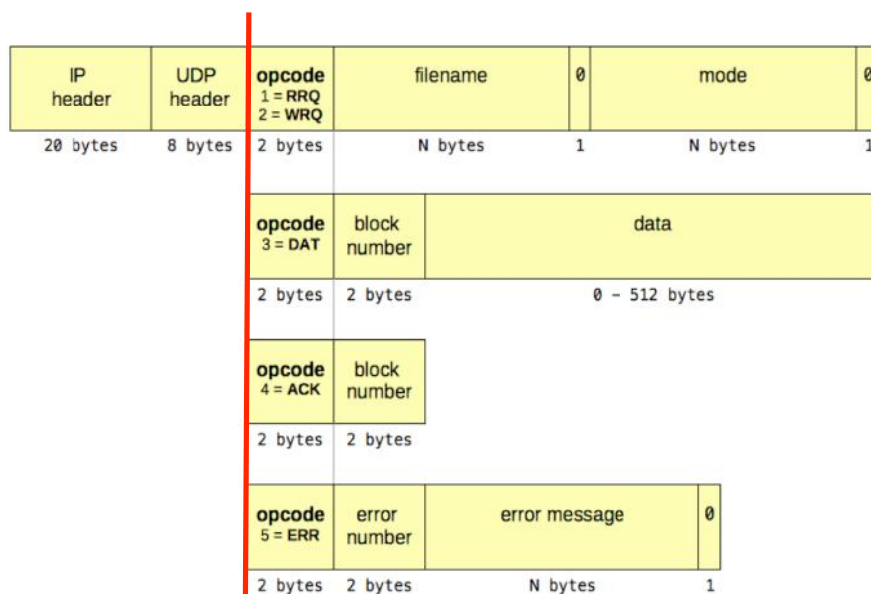
RRQ: Read Request - Opcode 1

WRQ: Write Request - Opcode 2

DAT: Data transfer - Opcode 3

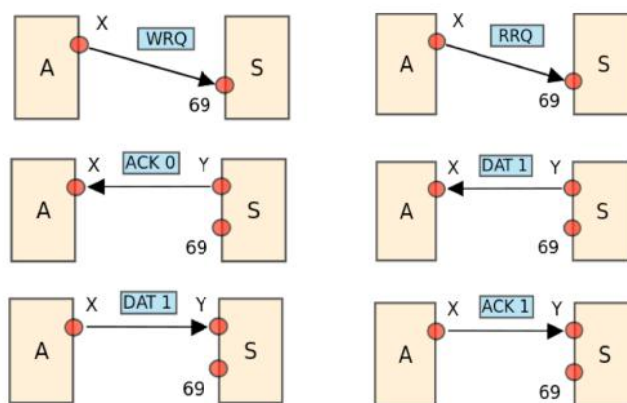
ACK: Acknowledge - Opcode 4

ERR: Error - Opcode 5



Formato dos pacotes/mensagens TFTP

- Este protocolo pertence à categoria de protocolos *stop-and-wait*: o cliente inicia o pedido de leitura (RRQ) ou de envio de um ficheiro (WRQ) e depois a transferência prossegue em blocos de 512 bytes de cada vez (DAT). Cada bloco de 512 bytes tem que ser reconhecido individualmente através de um pacote ACK. O bloco N só é transferido após o ter sido recebido o ACK para o bloco $N-1$. Ou seja, em cada instante só existe um bloco de dados a circular na rede.

(a) A envia um ficheiro para S (b) A recebe ficheiro vindo de S

NOTA: imagens tiradas da página da Wikipedia sobre o TFTP

De notar que um WRQ é reconhecido com um ACK numerado a partir de 0, ao passo que a resposta a um RRQ é um pacote DAT numerado a partir de 1.

- O receptor responde a um pacote DAT com blocknumber X com o correspondente pacote ACK X . Por seu turno, o emissor responde ao ACK X com o envio do DAT $X + 1$.

Tendo em conta que estamos a utilizar UDP, o emissor pode nunca chegar a receber o ACK. Isto pode acontecer devido a dois motivos: 1) O pacote de dados não chegou ao receptor, logo não pode ser enviado um ACK para um pacote que se desconhece; 2) O pacote DAT chegou e o receptor respondeu com ACK, mas foi este pacote que se extraviou. Seja como for, se não for recebido um ACK para o pacote X, então um temporizador do lado do emissor faz com que este reenvie o pacote DAT X.

- O servidor responde a um RRQ com um DAT 1 (`blocknumber` → 1), e a um WRQ com um ACK 0 (`blocknumber` → 0). Os pacotes RRQ e WRQ são enviados para o porto conhecido do servidor que, por omissão, é o porto 69. Porém, para a restante a comunicação o servidor utiliza um outro porto obtido aleatoriamente junto do sistema operativo local (*ephemeral port*). Os pacotes DAT 1 e ACK 0 são enviados a partir deste novo porto. Isto permite que o servidor continue a receber pedidos de transferência de ficheiros no porto 69 em simultâneo com a transferência de ficheiros em outros portos.
- Do lado do receptor do ficheiro (o cliente no caso de um RRQ, o servidor no caso de um WRQ), uma transferência termina assim que este detecta um bloco com dimensão inferior a 512 bytes. Se o tamanho do ficheiro for múltiplo de 512 bytes, o emissor deve terminar enviando um pacote com DAT com 0 bytes de dados.
- Ambas as partes, emissor e receptor, devem estar preparadas para que a contraparte deixe de responder (por exemplo, uma das partes pode abortar inesperadamente). No caso do receptor deixar de responder, a dada altura o emissor não vai receber um determinado ACK. Se for o emissor a não responder, o receptor não vai receber o último pacote com 0 bytes a sinalizar ao fim da transmissão do ficheiro. Ambas as partes devem cancelar uma transmissão que esteja inactiva há mais de 60s. Porém, este é um dos aspectos avançados cuja implementação é opcional, quer do lado do cliente, quer do lado do servidor.
- Como referido, a transferência de ficheiros é feita em blocos de 512 bytes reconhecidos à vez. Existe a opção de cliente e servidor negociarem um bloco de maior dimensão, o que permite tornar o protocolo mais eficiente. No entanto, neste projecto vamos ignorar este e os restantes aspectos relacionados com a negociação de parâmetros.
- O TFTP utiliza 16 bits para numerar bloco de dados, o que permite 65535 blocos (o primeiro DAT é sempre numerado a partir de 1, mas o primeiro ACK como resposta a um WRP é numerado a partir de 0). Inicialmente, e atendendo ao facto de cada bloco comportar no máximo 512 bytes, isto limitava os ficheiros a um máximo de 32MB. Hoje em dia este limite é ignorado o campo `blocknumber` volta a 0 após ter atingido o valor 65535.
- O protocolo suporta três modos transferência: `netascii`, `octet` ou `mail`. O modo `octet` sinaliza ao servidor uma transferência binária, sendo este o único modo a implementar neste projecto. Para

indicar este modo, o campo `mode` de um `RRQ` ou de um `WRQ` deve possuir a string binária `'octet\0'`, sendo aceite qualquer combinação de maiúsculas ou minúsculas. A string deve terminar com o caractere nulo, isto é, com o caractere com código 0 (`\x00` na notação hexadecimal utilizada pelo Python). Note que em Python 3 deve prefixar a string com `b` para indicar que se trata de uma string binária (ie, uma string do tipo `bytes`). A título de curiosidade, os restantes modos são indicados por qualquer combinação de maiúsculas e minúsculas de `'netsascii\0'` e `'mail\0'`.

- O campo `filename` presente nos pacotes `RRQ` e `WRQ` apenas deve conter caracteres ASCII "exibíveis" (*printable*). Consulte `string.printable`.
- O TFTP não fornece quaisquer mecanismos de segurança. Não existe forma de especificar um nome de utilizador nem uma palavra-passe. Por esse motivo, é habitual em Unix um servidor de TFTP correr com o UID do utilizador `nobody` e o GID do grupo `nogroup`, o que leva a que apenas ficheiros com permissões de leitura e escrita para o "resto do mundo" estejam acessíveis. Em sistemas Unix, é comum limitar o acesso aos ficheiros dentro de uma directoria pré-definida que, por norma, é `/tftpboot` ou `/var/lib/tftpboot`.
- O pacote de erros `ERR` é utilizado para informar a contraparte de um erro na transmissão. Por exemplo, o servidor responde com esta mensagem quando não consegue processar um `RRQ` ou um `WRQ`. Isto pode suceder se o cliente tentar ler um ficheiro que não existe ou para o qual não tem permissões. Neste caso o servidor envia um código de erro em `errornumber` e uma mensagem em ASCII (uma string binária, uma vez mais) com a `errormessage`. Erros durante a transmissão de um ficheiro também podem levar ao envio desta mensagem.

Neste projecto, um pacote `ERR` deve levar a que a transferência seja cancelada. Cliente e servidor devem exibir o código do erro em `errornumber` e a mensagem de erro em `errormessage`.

Caso ocorra um erro durante a transmissão, o receptor deve apagar o ficheiro que está a ser transferido (caso o tenha criado antes da transferência ter sido concluída).

- Se o cliente enviar um `RRQ` com `filename` vazio, o servidor pode responder com uma listagem da directoria de/para onde os ficheiros são transferidos. Esta listagem deve ser obtida com o comando `ls -Alh` e depois enviada em pacotes `DAT` para o cliente. O cliente solicita este serviço em modo interactivo através da invocação do comando `dir` (ver secção sobre o cliente).

CONTINUA --->

O CLIENTE

Utilização

O cliente deve ser invocado assim:

```
$ python3 client.py (get|put) [-p serv_port] server source_file [dest_file]
$ python3 client.py [-p serv_port] server
```

A primeira forma de utilização invoca o cliente em modo não-interactivo. O valor por omissão de `serv_port` é 69. O parâmetro `server` deve receber o nome do servidor, enquanto que `source_file` e `dest_file` representam, respectivamente, o caminho para o ficheiro a transferir, ficheiro esse que pode ser gravado na máquina que o recebe com o nome `dest_file`. Alternativamente, a invocação em modo não-interactivo pode também ser assim especificada:

```
$ python3 client.py get [-p serv_port] server remote_file [local_file]
$ python3 client.py put [-p serv_port] server local_file [remote_file]
```

Se o nome do servidor não existir, a mensagem a exibir deve ser:

```
Unknown server: <nome_do_servidor>.
```

Se a comunicação falhar, a mensagem a exibir deve ser:

```
Error reaching the server '<nome do servidor>' (<ip do servidor>).
```

A segunda forma, invoca o cliente em modo interactivo. Este cliente é um elemento **extra** do projecto. O cliente interactivo permite aceder à seguinte *prompt*:

```
Exchaging files with server '<nome do servidor>' (<ip do servidor>)
tftp client>
```

Quatro comandos obrigatórios e um opcional são reconhecidos pelo cliente:

<code>get</code> ficheiro_remoto [ficheiro_local]	Descarrega ficheiro_remoto e grava-o com esse nome ou com o nome ficheiro_local.
<code>put</code> ficheiro_local [ficheiro_remoto]	Envia ficheiro_local com esse nome ou com o nome ficheiro_remoto.
<code>quit</code>	Termina o cliente
<code>help</code>	Exibe a ajuda dos comandos
<code>dir</code>	Obtém uma listagem de ficheiros na directoria do servidor. Este comando, e consequentes modificações do protocolo e do servidor são opcionais.

Eis o exemplo de uma sessão interactiva entre cliente e servidor:

```
$ python3 cliente.py alberto.local
Unknown server: alberto.local

$ python3 cliente.py armando.local
Exchanging files with server 'armando.local' (192.168.12.1).

tftp client> dir
total 15192
-rw-rw-rw- 1 nobody nogroup 1,0M 14 Feb 13:46 fich1.txt
-rw-rw-rw- 1 nobody nogroup 4,4M 14 Feb 13:46 fich2.bin
-rw-rw-rw- 1 nobody nogroup 1,5M 14 Feb 13:47 fich3.bin
-rw-rw-rw- 1 nobody nogroup 512K 14 Feb 13:47 fichX.bin
-rw-rw-rw- 1 nobody nogroup 48K 14 Feb 13:47 fichY.dat

tftp client> get
Usage: get remotefile [localfile]

tftp client> get fich2.bi
File not found.

tftp client> get fich2.bin
Received file 'fich2.bin' 4608000 bytes.

tftp client> send data.zip
Unknown command: 'send'.

tftp client> help
Commands:
  get remote_file [local_file] - get a file from server and save it as local_file
  put local_file [remote_file] - send a file to server and store it as remote_file
  dir                          - obtain a listing of remote files
  quit                         - exit TFTP client

tftp client> put data.zip
Sent file 'data.zip' 1048576 bytes

tftp client> quit
Exiting TFTP client.
Goodbye!
```

Qualquer erro na comunicação deverá levar ao encerramento da aplicação com exibição de uma mensagem de erro:

```
tftp client> get fich2.bin
Server not responding. Exiting.
```

Implementação

De um modo geral, quando iniciado em modo interactivo o seu cliente deverá levar a cabo as seguintes tarefas:

1. Obter o IP do servidor a partir do nome, caso tenha sido introduzido um nome. Se não tiver sucesso, exibe a mensagem de erro apropriada e termina.
2. Exibir a prompt e aguardar pela introdução de um comando.
3. Caso o comando seja:

PUT (enviar um ficheiro)

- 3.1 Obter caminho para o ficheiro a enviar a partir do comando. Deve verificar se o utilizador especificou um apenas um caminho (`local_file`) ou dois caminhos (`local_file` e `remote_file`).
- 3.2 Abrir o ficheiro local para leitura em modo binário. Terminar com mensagem de erro se não tiver sucesso.
- 3.3 Criar um socket do tipo datagrama para comunicar com o servidor. Terminar com mensagem de erro se não tiver sucesso.
- 3.4 Enviar ficheiro em blocos de 512 bytes de acordo com o protocolo TFTP.
Deve enviar o ficheiro para o porto de onde o servidor respondeu que provavelmente não é o 69. Deve estar preparado para a eventualidade de receber uma mensagem de erro (**ERR**) do servidor. Exibir mensagem "Server not responding..." se estiver 60s sem obter resposta (**ACK**) do servidor.
- 3.5 Fechar ficheiro quando terminada a transferência.
- 3.6 Notificar o utilizador do resultado da transferência e quantos bytes foram enviados.

GET (receber um ficheiro)

- 3.7 Obter nome do ficheiro a receber a partir do comando. Deve verificar se o utilizador especificou um apenas um caminho (`remote_file`) ou dois caminhos (`remote_file` e `local_file`).
- 3.8 Abrir ficheiro local para escrita em modo binário. Terminar com mensagem de erro se não tiver sucesso.
- 3.9 Criar um socket como em PUT.
- 3.10 Receber ficheiro em blocos de 512 bytes de acordo com o protocolo TFTP.
Como em PUT, deve também ter em atenção o porto do servidor por onde é enviado o primeiro **DAT**.
Exibir mensagem "Server not responding..." se passarem 60s entre dois blocos **DAT** consecutivos. Deve estar preparado para eventuais mensagens de erro do lado do servidor. Caso ocorra um erro durante a transferência do ficheiro, deve abortar e apagar o ficheiro criado para escrita.
- 3.11 Fechar ficheiro quando terminada a transferência.
- 3.12 Notificar o utilizador do resultado da transferência e quantos bytes foram recebidos.

DIR (listagem)

- 3.13 Semelhante a GET mas solicita um ficheiro sem nome (`filename == b''`).
4. Repetir passos 2-4 enquanto o utilizador não desistir ou o servidor não enviar uma mensagem de erro.

O cliente deve obedecer ao protocolo e não deve aceitar mensagens inválidas de acordo com o mesmo. Por exemplo, não é suposto cliente receber pacotes **RRQ** nem **WRQ**. Também não é suposto receber **ACK** durante

uma transferência iniciada com **RRQ**. Nestes casos a transferência actual deve ser abortada.

Uma complexidade adicional está relacionada com o alinhamento dos bytes em arquitecturas diferentes. Por exemplo, para um CPU da Motorola um número de 16 bits com o valor 3 é representado em memória como 0x0003 (hexadecimal). Em máquinas Intel, os bytes estão invertidos, ou seja, o mesmo número é representado como 0x0300 (byte menos significativo em primeiro lugar). Diz-se que a primeira arquitectura é *big endian* ao passo que a segunda é *little endian*. Neste projecto necessita de lidar com os seguintes valores de 16 bits: opcode, blocknumber e errornumber. À saída do cliente, quando constrói os pacotes apropriados, deve converter estes valores para o formato canónico da rede (*big endian*) com a função `socket.htons` (*host to network short*, sendo que *short*, na linguagem C, é inteiro de 16 bits). Para ler estes campos nos pacotes enviados pelo servidor, deve utilizar a função `socket.ntohs` (*network to host short*) para fazer a conversão inversa. Pode encontrar explicações adicionais nas referências [3] e [4].

Como alternativa às funções `socket.ntohs` & `Ca Lda.`, pode recorrer ao módulo `struct` para gerar os pacotes TFTP já com o alinhamento de bytes certo para enviar pela rede. "Oficialmente", este módulo permite formatar valores dos tipos de dados nativos do Python para os tipos de dados primitivos da linguagem C. Mas no caso concreto em questão podemos ignorar esses aspectos da linguagem C e utilizar as funções deste módulo para gerar a sequência de bytes correspondente ao pacote que pretendemos enviar. Por exemplo, suponha que pretende enviar um pacote **RRQ** a solicitar o ficheiro 'xpto.bin' em modo `octet`. O pacote **RRQ** pode ser gerado da seguinte forma:

```
>>> import struct
>>> opcode = 1                                # RRQ
>>> filename = input("Nome do ficheiro? ").encode()    # assume 'xpto.bin'
>>> filename += b'\0'
>>> mode = b'octet\0'
>>> pack_fmt = '!h{}s6s'.format(len(filename))        # produz '!h9s6s'
>>> # o '!' garante o alinhamento binário certo (big endian) para a rede

>>> packet = struct.pack(pack_fmt, opcode, filename, mode)
>>> packet
b'\x00\x01xpto.bin\x00octet\x00'
# ... agora é uma questão de enviar o pacote, mas isso fica para o projecto ...
```

No lado do receptor, a função `struct.unpack` permite desempacotar o pacote para obter os campos novamente. Note que para "desempacotar" o filename e o mode necessita de localizar o caractere nulo `b'\0'` em cada uma das strings binárias. No caso do filename, o caractere nulo deve ser procurado a partir do terceiro byte (inclusive), já que os dois primeiros estão reservados para o opcode.

Para trabalhar com sockets *datagrama* (`SOCK_DGRAM`) sugere-se que consulte a secção 11.3 da referência [5]. Esta secção, bem como a anterior, podem também ser úteis para implementar o servidor.