Implémentation du protocole TFTP

Kevin Estalella, Federico Lerda, Federico Pfeiffer

22 décembre 2016

Résumé

Ce rapport décrit l'implémentation du protocole TFTP dans le langage python.

Table des matières

1 Présentation

Le projet consiste en l'implémentation du protocole TFTP. Ce protocole est une version light du protocole FTP. Le but de ce projet est d'implémenter une version simplifiée de TFTP en utilisant les méthodes vues en cours. Le protocole est implémenté en se basant sur la RFC 1350.

2 Protocole TFTP

2.1 Vue d'ensemble

Le protocole TFTP est un protocole servant à envoyer / recevoir des fichiers entre un client et un serveur. Les deux parties peuvent se comporter comme émetteur et récepteur. La transmission d'un fichier débute par une requête de la part du client, dans le but de demander à envoyer un fichier (requête représentée par un paquet WRQ), ou dans le but de demander à recevoir un fichier (requête représentée par un paquet RRQ). Si le serveur accepte, le transfert de fichier commence.

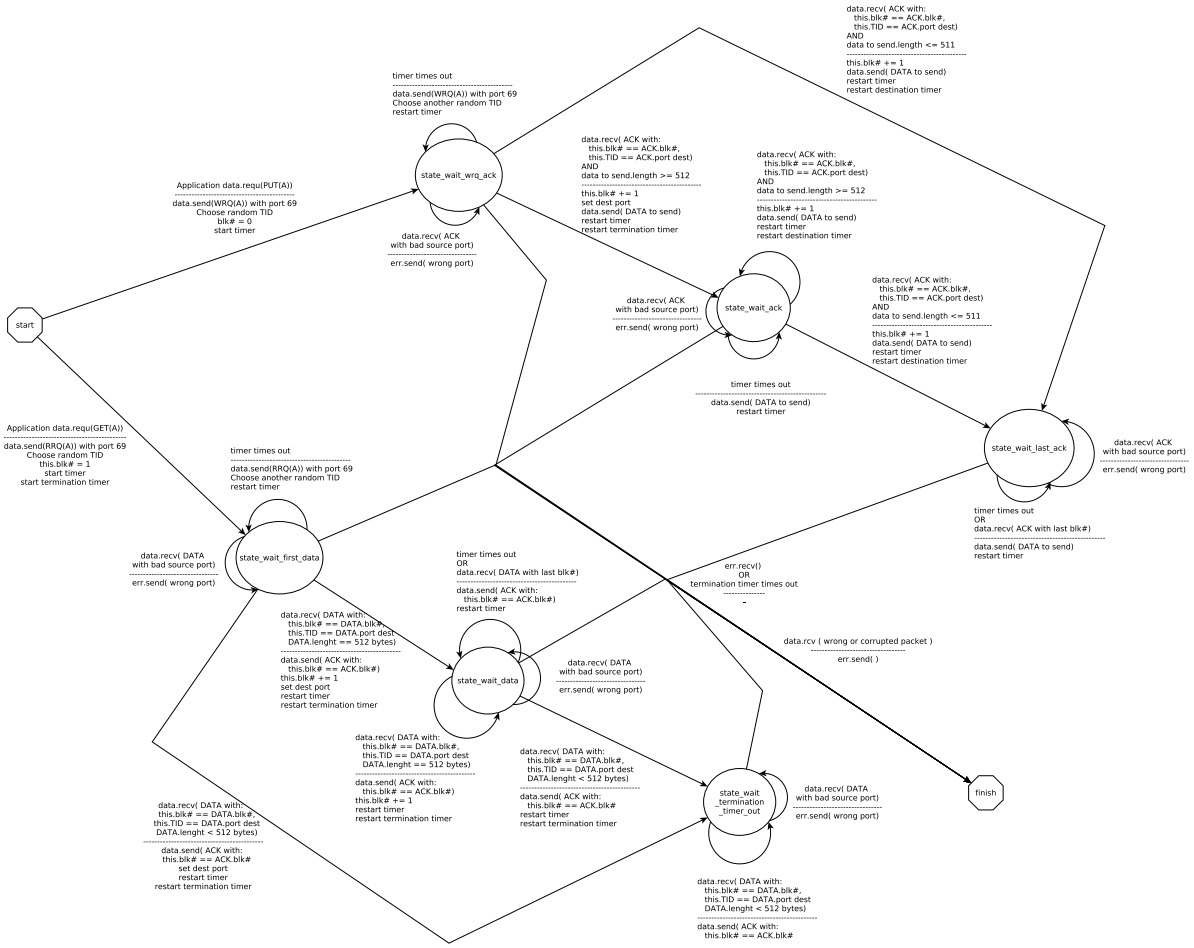
Les fichiers sont transmis par paquets de données d'une taille de 512 Octets. Un paquet de données contenant moins de 512 Octets signifie qu'il s'agit le dernier paquet.

Chaque paquet contient un numéro de paquet et est acquitté par le destinataire par un paquet ACK ; l'émetteur d'un paquet de données doit attendre la réception d'un ACK avant d'envoyer le paquet de données suivant. En cas d'erreur, un paquet ERR est envoyé et le transfert est annulé et terminé.

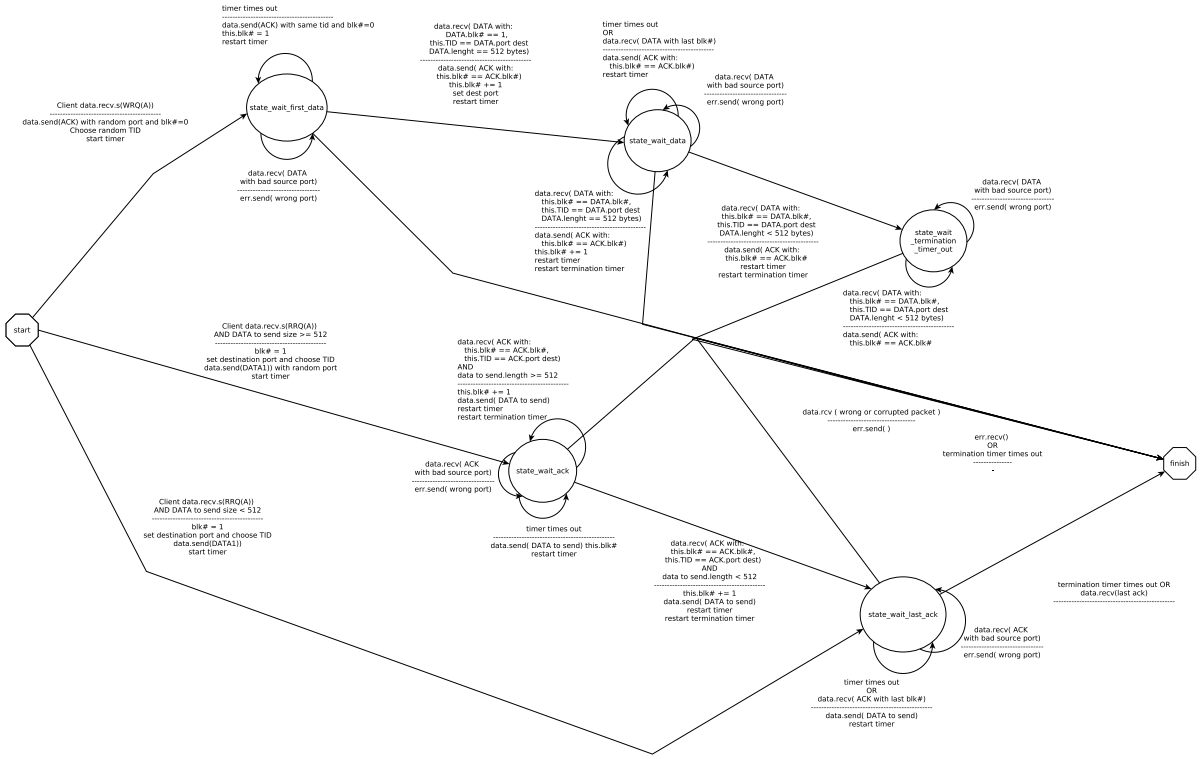
2.2 Machines d'états

Les machines d'états suivantes ont été créés en se basant sur le descriptif de la RFC 1350.

2.2.1 Client


start
Application data.requ(PUT(A))
data.send(WRQ(A)) with port 69
Choose random TID
blk# = 0
start timer
state_wait_wrq_ack
timer times out
data.send(WRQ(A)) with port 69
Choose another random TID
restart timer
data.recv(ACK with bad source port)
err.send(wrong port)
data.recv(ACK with:
this.blk# == ACK.blk#,
this.TID == ACK.port dest)
AND
data to send.length >= 512
this.blk# += 1
set dest port
data.send(DATA to send)
restart timer
restart termination timer
data.recv(ACK with:
this.blk# == ACK.blk#,
this.TID == ACK.port dest)
AND
data to send.length >= 512
this.blk# += 1
data.send(DATA to send)
restart timer
restart destination timer
state_wait_ack
data.recv(ACK with bad source port)
err.send(wrong port)
data.recv(ACK with:
this.blk# == ACK.blk#,
this.TID == ACK.port dest)
AND
data to send.length <= 511
this.blk# += 1
data.send(DATA to send)
restart timer
restart destination timer
timer times out
data.send(DATA to send)
restart timer
state_wait_last_ack
data.recv(ACK with bad source port)
err.send(wrong port)
timer times out
OR
data.recv(ACK with last blk#)
data.send(DATA to send)
restart timer
Application data.requ(GET(A))
data.send(RRQ(A)) with port 69
Choose random TID
this.blk# = 1
start timer
start termination timer
timer times out
data.send(RRQ(A)) with port 69
Choose another random TID
restart timer
state_wait_first_data
data.recv(DATA with bad source port)
err.send(wrong port)
timer times out
OR
data.recv(DATA with last blk#)
data.send(ACK with:
this.blk# == ACK.blk#)
restart timer
err.recv()
OR
termination timer times out
-
data.recv(DATA with:
this.blk# == DATA.blk#,
this.TID == DATA.port dest
DATA.lenght == 512 bytes)
data.send(ACK with:
this.blk# == ACK.blk#)
this.blk# += 1
set dest port
restart timer
restart termination timer
state_wait_data
data.recv(DATA with bad source port)
err.send(wrong port)
data.rcv (wrong or corrupted packet)
err.send( )
data.recv(DATA with:
this.blk# == DATA.blk#,
this.TID == DATA.port dest
DATA.lenght == 512 bytes)
data.send(ACK with:
this.blk# == ACK.blk#)
this.blk# += 1
restart timer
restart termination timer
data.recv(DATA with:
this.blk# == DATA.blk#,
this.TID == DATA.port dest
DATA.lenght < 512 bytes)
data.send(ACK with:
this.blk# == ACK.blk#
restart timer
restart termination timer
data.recv(DATA with:
this.blk# == DATA.blk#,
this.TID == DATA.port dest
DATA.lenght < 512 bytes)
data.send(ACK with:
this.blk# == ACK.blk#
set dest port
restart timer
restart termination timer
state_wait
_termination
_timer_out
data.recv(DATA with bad source port)
err.send(wrong port)
data.recv(DATA with:
this.blk# == DATA.blk#,
this.TID == DATA.port dest
DATA.lenght < 512 bytes)
data.send(ACK with:
this.blk# == ACK.blk#
finish

2.2.2 Serveur

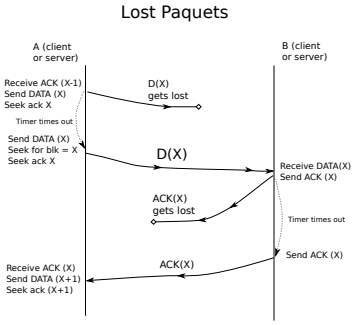


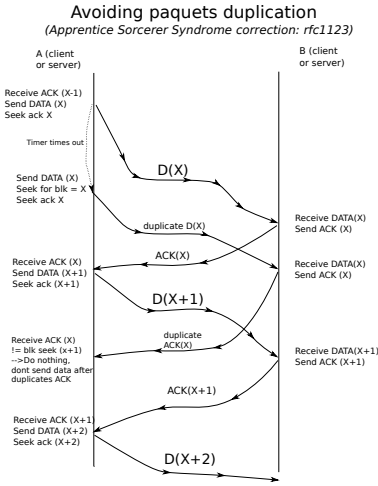
2.3 Cas d'erreur

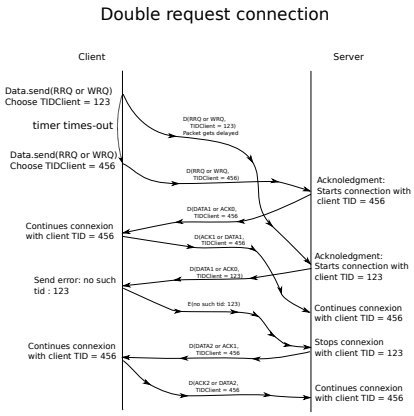
Les cas d'erreurs suivants ont été analysés dans le but de tester la robustesse du protocole. Le premier exemple décrit la manière dont le protocole se comporte en cas de perte de paquets (coté client ou serveur). Les paquets sont simplement renvoyés en cas de timeout.

Le deuxième cas (associé à ce qui est appelé syndrome de l'apprenti sorcier), décrit les erreurs qui peuvent survenir lors de délais dans la transmission des paquets. Lorsque le délai est suffisamment grand, le paquet retardé est à nouveau envoyé, ce qui crée une duplication des paquets. Il faut s'assurer que cette duplication de paquets ne se propage pas pendant toute la durée du transfert. Ceci est résolu de la manière suivante : lorsqu'on reçoit un ACK concernant le paquet de données qu'on vient d'envoyer, on envoie le paquet de données suivant. Cependant, si on reçoit à nouveau le même ACK, on ne renvoie rien.

Le dernier cas illustré décrit le cas où un paquet de requête (RRQ ou WRQ) a été retardé. Suite au retard, un deuxième paquet de requête est envoyé, ce qui pourrait conduire à une double connexion entre le client et le serveur. Pour éviter ceci et d'envoyer deux fois le même fichier, lorsqu'un transfert de fichier débute, le client et le serveur choisissent un identifiant à utiliser tout le long du transfert. (identifiant inscrit comme ports UDP). Lorsque le client reçoit l'accord du serveur pour commencer le transfert de fichier, le client va enregistrer l'ID du serveur que celui-ci a envoyé pour le transfert de fichier. Lorsque le client reçoit le deuxième accord du serveur avec un ID différent, le client ne va pas initier une deuxième connexion avec le serveur car il est déjà avec une connexion avec celui-ci. Il n'y aura donc pas de double connexions.







3 Architecture du programme

L'architecture de notre implémentation a été définie comme suit. Deux fichiers client.py et server.py qui contiendront l'algorithme définit dans les machines d'états.
Partant du principe que la manière de transférer les fichiers est la même pour le serveur et le client, nous avons définit un fichier utils.py qui contiendra les méthodes pour envoyer et recevoir des fichiers. Ces méthodes sont utilisées par le client et le serveur.
Enfin, un fichier nommé packet.py, également utilisé par le serveur et le client, contient toutes les fonctions nécessaires pour encoder et décoder les paquets TFTP.

utils.py:

state_wait_ack()

state_wait_last_ack()

close_and_exit()

state_wait_data()

state_wait_termination_timer_out()

Enum States

Enum AppRq

client.py:

send_request()

state_wait_wrq_ack()

state_wait_first_data()

exec_client_state()

server.py:

parser()

state_wait_first_data()

listen()

exec_server_state()

packet.py:

Enum : OP_CODE
Enum : ERR_CODE

BinaryString : build_packet_rrq(Filename, mode)

BinaryString : build_packet_wrq(Filename, mode)

BinaryString : build_packet_data(Block_num, data)

BinaryString : build_packet_ack(Block_num)

BinaryString : build_packet_err(Err_code)

For RRQ and WRQ
Op_code, File_name, Mode : decode_packet(BinaryString)

For ACK
Op_code, Block_num : decode_packet(BinaryString)

For DATA and ERR
Op_code, Block_num, Msg : decode_packet(BinaryString)