Annexe

AMEDRO Brian, BODNARTCHOUK Vladimir, ROBITZER Vincent

26 juin 2005

Table des matières

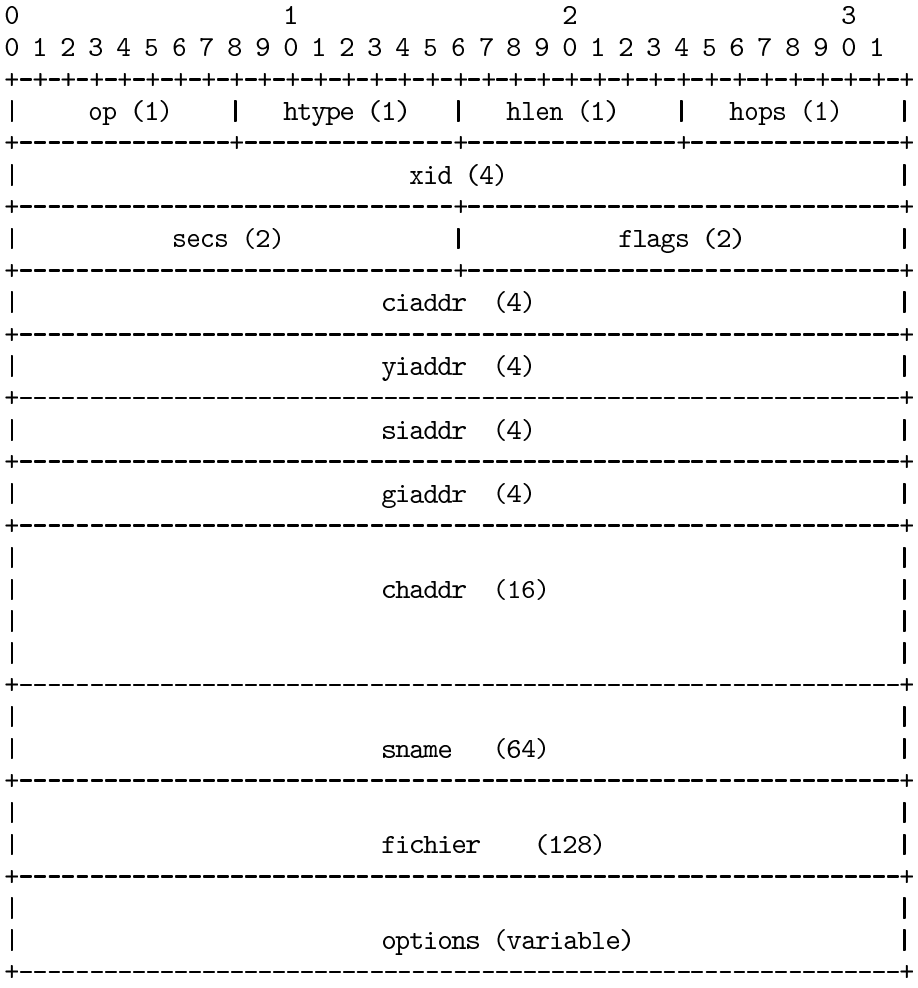
1 DHCP **5**
1.1 Format d'un message DHCP 5
1.2 Champs d'un message DHCP 6
1.3 Options de DHCP 6

2 Code source du prototype SDHCP **13**

Chapitre 1

DHCP

1.1 Format d'un message DHCP



1.2 Champs d'un message DHCP

CHAMP	OCTETS	DESCRIPTION
op	1	Code opération du message / type du message. 1 = BOOTREQUEST, 2 = BOOTREPLY
htype	1	Adresse matérielle, voir la section ARP dans le RFC "Assigned Numbers" ; par ex., '1' = Ethernet 10Mb.
hlen	1	Longueur de l'adresse matérielle (par ex. '6' for Ethernet 10Mb).
hops	1	Mis à zéro par le client, utilisé de manière optionnelle par les agents de relais quand on démarre via un agent de relais.
xid	4	Identifiant de transaction, un nombre aléatoire choisi par le client, utilisé par le client et le serveur pour associer les messages et les réponses entre un client et un serveur.
secs	2	Rempli par le client, les secondes s'écoulent depuis le processus d'acquisition ou de renouvellement d'adresse du client.
flags	2	Drapeaux.
ciaddr	4	Adresse IP des clients, rempli seulement si le client est dans un état AFFECTÉ, RENOUVELLEMENT ou REAFFECTATION et peut répondre aux requêtes ARP.
yiaddr	4	'votre' (client) adresse IP.
siaddr	4	Adresse IP du prochain serveur à utiliser pour le processus de démarrage ; retournée par le serveur dans DHCPOFFER et DHCPACK.
giaddr	4	Adresse IP de l'agent de relais, utilisée pour démarrer via un agent de relais.
chaddr	16	Adresse matérielle des clients.
sname	64	Nom d'hôte du serveur optionnel, chaîne de caractères terminée par un caractère nul.
fichier	128	Nom du fichier de démarrage, chaîne terminée par un caractère nul ; nom "generic" ou nul dans le DHCPDISCOVER, nom du répertoire explicite dans DHCPOFFER.
options	var	Champ de paramètres optionnels. Voir le document d'options pour une liste des options définies.

TAB. 1.1 – description des champs dans un message DHCP

1.3 Options de DHCP

option subnet-mask ip-address ;

L'option subnet-mask spécifie le masque de sous-réseau du client tel que défini dans la RFC 950 (RFC = Request For Comments). Si aucune option subnet-mask n'est fournie dans la portée du bloc, dhcpd utilisera en dernier recours le masque de la déclaration de sous-réseau auquel appartient l'adresse en cours d'attribution. Cependant, toute déclaration d'option subnet-mask remplacera le masque de sous-réseau spécifié dans la déclaration de sous-réseau.

option time-offset int32 ;

L'option time-offset spécifie le décalage en secondes pour le client de sous-réseau par rapport aux coordonnées de temps universelles (UTC : Universal Time Coordinates).

option routers ip-address [, ip-address...] ;

L'option routers spécifie une liste d'adresses IP de routeurs qui sont sur le sous-réseau du client. Les routeurs doivent être mentionnés par ordre de préférence.

option time-servers ip-address [, ip-address...] ;

L'option time-server spécifie une liste de serveurs de temps de type RFC 868 disponibles pour le client. Les serveurs doivent être spécifiés par ordre de préférence.

option ien116-name-servers ip-address [, ip-address...] ;

L'option ien116-name-servers spécifie une liste des serveurs de nom de type IEN 116 disponibles pour le client. Les serveurs doivent être spécifiés par ordre de préférence.

option domain-name-servers ip-address [, ip-address...] ;

L'option domain-name-servers spécifie une liste de serveurs de noms de de domaines (DNS : Domain Name System) (STD 13, RFC 1035) disponibles pour les clients. Les serveurs doivent être spécifiés par ordre de préférence.

option log-servers ip-address [, ip-address...] ;

L'option log-server spécifie une liste de serveurs MIT-LCS UDP d'enregistrement des logs disponibles pour le client. Les serveurs doivent être spécifiés par ordre de préférence.

option cookie-servers ip-address [, ip-address...] ;

L'option cookie-servers spécifie une liste de serveurs de cookies de type RFC 865 disponibles pour le client. Les serveurs doivent être spécifiés par ordre de préférence.

option lpr-servers ip-address [, ip-address...] ;

l'options lpr-servers spécifie une liste de serveurs d'impression de type RFC 1179 disponibles pour le client. Les serveurs doivent être spécifiés par ordre de préférence.

option impress-servers ip-address [, ip-address...] ;

L'option impress-servers spécifie une liste de serveur Imagen Impress disponibles pour le client. Les serveurs doivent être spécifiés par ordre de préférence.

option resource-location-servers ip-address [, ip-address...] ;

L'option resource-location-servers spécifie une liste de serveurs de localisation de ressource disponibles pour le client. Les serveurs doivent être spécifiés par ordre de préférence.

option host-name string ;

Cette option spécifie le nom du client. Le nom peut être entièrement qualifié avec le nom de domaine local, mais il est préférable d'utiliser l'option domain-name pour spécifier le nom de domaine. Voir la RFC 1035 pour les restrictions sur le jeu de caractères.

option boot-size uint16 ;

Cette option spécifie la longueur en blocs de 512 octets de l'image de démarrage par défaut du client.

option merit-dump string ;

Cette option spécifie le nom complet du fichier (avec le chemin) dans lequel le « core » (image mémoire) du client doit être enregistré dans le cas où ce dernier planterait. Le chemin est une chaîne de caractères NVT ASCII.

option domain-name string ;

Cette option spécifie, le nom de domaine que le client doit utiliser lorsqu'il résout des noms d'hôte via les serveurs de noms DNS

option swap-server ip-address ;

Cette option spécifie l'adresse IP du serveur de swap (pagination sur disque) du client.

option root-path string ;

Cette option spécifie le nom complet de fichier (avec le chemin) qui contient le disque racine du client (root disk). Le nom de fichier est une chaîne de caractères NVT ASCII.

option ip-forwarding flag ;

Cette option spécifie si le client doit configurer sa couche IP pour la propagation de paquets. La valeur 1 (resp. 0) active (resp. désactive) l'IP forwarding.

option non-local-source-routing flag ;

Cette option spécifie si un client doit configurer sa couche IP pour autoriser la propagation des datagrammes avec une route source non-locale (voir la section 3.3.5 de [4] pour une discussion de ce sujet). La valeur 1 (resp. 0) active (resp. désactive) la propagation de tels datagrammes.

option policy-filter ip-address ip-address [, ip-address ip-address...] ;

Cette option spécifie la politique de filtrage pour les routes sources non-locales. Les filtres consistent en une liste d'adresses IP et de masques qui spécifie les paires destination/masque pour le filtrage des entrées en fonction de la route source.

Tout datagramme source routé dont le prochain saut d'adresse ne correspond à aucun filtre sera rejeté par le client.

Voir STD 3 (RFC1122) pour plus d'informations.

option max-dgram-reassembly uint16 ;

Cette option spécifie la taille maximale des datagrammes que le client doit se préparer à réassembler. La valeur minimale légale est 576.

option default-ip-ttl uint8 ;

L'option spécifie la valeur par défaut du TTL (Time-To-Live, temps à vivre) que le client doit utiliser pour les datagrammes en sortie.

option path-mtu-aging-timeout uint32 ;

Cette option spécifie la durée d'expiration (timeout) à utiliser quand on vieillit l'unité de transfert maximal d'un chemin (Path Maximum Transfert Unit) découverte par le mécanisme défini dans la RFC 1191.

option path-mtu-plateau-table uint16 [, uint16...] ;

L'option spécifie un tableau de MTU (Maximal Transfert Unit, unité maximale de transfert) à essayer lors de l'opération de découverte des MTU des chemins telle que définie dans la RFC 1191. La table est une liste d'entier non-signés de 16 bits, ordonnés du plus petit au plus grand. La valeur minimale de MTU est 68.

option interface-mtu uint16 ;

L'option spécifie la MTU à utiliser pour cette interface. La valeur minimum légale est 68.

option all-subnets-local flag ;

Cette option spécifie si le client peut supposer que tous les sous-réseaux du réseau IP auquel il est connecté utilisent la même MTU que celui auquel il est directement connecté. La valeur 1 indique que tous les sous-réseaux partagent la même MTU. La valeur 0 indique que le client doit supposer que certains sous-réseaux du réseau ont des MTU plus petites.

option broadcast-address ip-address ;

Cette option spécifie l'adresse de diffusion à utiliser pour le sous-réseau du client. Les valeurs légale sont spécifiées dans la section 3.2.1.3 du STD 3 (RFC 1122).

option perform-mask-discovery flag ;

Cette option spécifie si le client doit découvrir le masque de sous-réseau en utilisant le protocole ICMP. La valeur 1 (resp. 0) indique que le client doit (resp. ne doit pas) faire la découverte de masque.

option mask-supplier flag ;

Cette option spécifie si le client doit répondre aux requêtes de masque de sous-réseau en utilisant ICMP. La valeur 1 (resp. 0) indique que le client doit (resp. ne doit pas) répondre.

option router-discovery flag ;

Cette option spécifie si le client doit solliciter les routeurs en utilisant le mécanisme de découverte des routeurs défini dans la RFC 1256. La valeur 1 (resp. 0) indique que le client doit (resp. ne doit pas) faire une découverte des routeurs.

option router-solicitation-address ip-address ;

Cette option spécifie l'adresse à laquelle le client doit transmettre ses requêtes de sollicitations de routeurs.

option static-routes ip-address ip-address [, ip-address ip-address...] ;

Cette option spécifie une liste de routes statiques que le client doit installer dans son cache de routage. Si plusieurs routes menant à la même destination sont spécifiées, elles doivent être mentionnées dans l'ordre décroissant des priorités.

Les routes consistent en une liste de paires d'adresse IP. La première adresse IP est l'adresse de destination et la seconde est l'adresse du routeur pour cette destination.

La route par défaut (0.0.0.0) est une destination illégale pour une route statique. Pour spécifier une route par défaut, utilisez l'option routers.

option trailer-encapsulation flag ;

Cette option spécifie si le client doit négocier l'utilisation de trailers (RFC 893 [14]) lors de l'utilisation du protocole ARP (Address Resolution Protocol). La valeur 1 (resp. 0) indique que le client doit (resp. ne doit pas) essayer d'utiliser les trailers.

option arp-cache-timeout uint32 ;

Cette option spécifie la durée d'expiration en secondes des entrées dans le cache ARP.

option ieee802-3-encapsulation flag ;

Cette option spécifie si le client doit utiliser l'encapsulation Ethernet Version 2 (RFC 894) ou IEEE 802.3 (RFC 1042) si son interface est de type Ehernet. La valeur 0 (resp. 1) indique que le client doit utiliser l'encapsulation RFC 894 (resp. 1042).

option default-tcp-ttl uint8 ;

Cette option spécifie la valeur par défaut pour le TTL (time-to-live) qu'un client doit utiliser lorsqu'il envoie des paquets TCP. La valeur minimale est 1.

option tcp-keepalive-interval uint32 ;

Cette option spécifie l'intervalle (en secondes) que le client TCP doit attendre avant d'envoyer un message « keepalive » sur une connexion TCP. Le temps est spécifié sous forme d'un entier non-signé de 32 bits. La valeur 0 indique que le client ne doit pas générer de message keepalive sur les connexions à moins que ce ne soit spécifiquement demandé par une application.

option tcp-keepalive-garbage flag ;

Cette option spécifie si le client doit envoyer des messages TCP keepalive avec un octet de bourrage pour la compatibilité avec les vieilles implémentations. La valeur 1 (resp. 0) indique que l'octet de bourrage doit (resp. ne doit pas) être envoyé.

option nis-domain string ;

Cette option spécifie le nom du domaine NIS (Sun Network Information Services) du client. Le domaine est une chaîne de caractères NVT ASCII.

option nis-servers ip-address [, ip-address...] ;

Cette option spécifie une liste d'adresse IP de serveur NIS disponibles pour le client. Les serveurs doivent être mentionnés par ordre de préférence.

option ntp-servers ip-address [, ip-address...] ;

Cette option spécifie une liste de d'adresse IP de serveurs NTP (RFC 1035) disponibles pour le client. Les serveurs doivent être mentionnés par ordre de préférence.

option netbios-name-servers ip-address [, ip-address...] ;

L'option netbios-name-servers spécifie une liste de serveurs NetBIOS (NBNS : NetBIOS Name System) RFC 1001/1002 mentionnés par ordre préférence. Le service de nom NetBIOS est aussi communément appelé WINS. Les serveurs WINS peuvent être spécifiés en utilisant l'option netbios-name-servers.

option netbios-dd-server ip-address [, ip-address...] ;

L'option netbios-dd-server spécifie une liste de serveur de distribution de datagramme NetBIOS (NBDD : NetBIOS Datagram Distribution) RFC 1001/1002. Les serveurs doivent être mentionnés par ordre de préférence.

option netbios-node-type uint8 ;

L'option netbios-node-type autorise la configuration des clients NetBIOS sur TCP/IP comme décrit dans la RFC 1001/1002. La valeur est spécifiée sous forme d'un unique octet qui identifie le type de client.

Les type possibles de noeuds sont :

1 B-node : Broadcast - no WINS (diffusion - pas WINS) 2 P-node : Peer - WINS only (point-à-point - WINS seulement) 4 M-node : Mixed - broadcast, then WINS (mélangé - diffusion puis WINS) 8 H-node : Hybrid - WINS, then broadcast (hybride - WINS, puis diffusion)

option netbios-scope string ;

L'option netbios-scope spécifie la portée des paramètres de NetBIOS sur TCP/IP pour les clients, comme indiquée dans la RFC 1001/1002. Voir la RFC1001, la RFC1002, et la RFC1035 pour les limitations du jeu de caractères.

option font-servers ip-address [, ip-address...] ;

Cette option spécifie une liste de serveurs de polices X Window disponibles pour les clients. Les serveurs doivent être mentionnés par ordre de préférence.

option x-display-manager ip-address [, ip-address...] ;

Cette option spécifie une liste de systèmes qui sont en train de faire fonctionner un gestionnaire de fenêtre X Window et qui sont disponibles pour le client. Les adresses doivent être spécifiées par ordre de préférence.

option dhcp-client-identifier data-string ;

Cette option peut être utilisée pour spécifier l'identifiant DHCP du client dans une déclaration d'hôte, de telle sorte que dhcpd puisse trouver l'enregistrement d'hôte en le mettant en correspondance avec l'identifiant du client.

option nisplus-domain string ;

Cette option spécifie le nom de domaine NIS+ du client. Le domaine est une chaîne de caractères NVT ASCII.

option nisplus-servers ip-address [, ip-address...] ;

Cette option spécifie une liste d'adresse IP de serveurs NIS+ disponibles pour le client. Les serveurs doivent être mentionnés par ordre de préférence.

option tftp-server-name string ;

Cette option est utilisée pour identifier un serveur TFTP et, si le client supporte cette option, elle devrait avoir le même effet que la déclaration server-name. Les clients BOOTP ne supportent probablement pas cette option. Certains clients DHCP la supporteront, et d'autres la voudront obligatoirement.

option bootfile-name string ;

Cette option est utilisée pour identifier le fichier de démarrage. Si le client supporte cette option, elle devrait avoir le même effet que la déclaration filename. Les clients BOOTP ne supportent probablement pas cette option. Certains clients DHCP la supporteront, et d'autres la voudront obligatoirement.

option mobile-ip-home-agent ip-address [, ip-address...] ;

Cette option spécifie une liste d'adresse IP d'agents mobiles. Les agents doivent être mentionnés par ordre de préférence, bien que normalement il n'y en aura qu'un seul.

option smtp-server ip-address [, ip-address...] ;

Cette option spécifie une liste d'adresse de serveur SMTP disponibles pour le client. Les serveurs doivent être mentionnés par ordre de préférence.

option pop-server ip-address [, ip-address...] ;

Cette option spécifie une liste de serveurs POP3 disponibles pour le client. Les serveurs doivent être mentionné par ordre de préférence.

option nntp-server ip-address [, ip-address...] ;

Cette option spécifie une liste de serveurs NNTP disponibles pour le client. Les serveurs doivent être mentionné par ordre de préférence.

option www-server ip-address [, ip-address...] ;

Cette option spécifie une liste de serveurs WWW (World Wide Web i.e. HTTP) disponibles pour le client. Les serveurs doivent être mentionnés par ordre de préférence.

option finger-server ip-address [, ip-address...] ;

Cette option spécifie une liste de serveurs Finger disponibles pour le client. Les serveurs doivent être mentionnés par ordre de préférence.

option irc-server ip-address [, ip-address...] ;

Cette option spécifie une liste de serveurs IRC disponibles pour le client. Les serveurs doivent être mentionnés par ordre de préférence.

option streettalk-server ip-address [, ip-address...] ;

Cette option spécifie une liste de serveurs StreetTalk disponibles pour le client. Les serveurs doivent être mentionnés par ordre de préférence.

option streetalk-directory-assistance-server ip-address [, ip-address...] ;

Cette option spécifie une liste de serveurs StreetTalk Directory Assistance (STDA) disponibles pour le client. Les serveurs doivent être mentionnés par ordre de préférence.

Chapitre 2

Code source du prototype SDHCP

```
/*
 * Mécanismes de configuration automatique d'une interface réseau
 * Travail d'étude 2005 - Université de Nice Sophia-Antipolis
 * AMEDRO Brian, BORDNARTCHOUK Vladimir, ROBITZER Vincent
 *
 * Prototype d'implémentation du protocole S-DHCP
 *
 * packet.h
 * Développement : AMEDRO Brian
 *
 */

int packet_get_option( unsigned char *packet, char option, int *len, char *data );

int paket_init( char *packet, char op, char flag, short xid );

int packet_add( char *packet, char option, short len, char *data );

int packet_close( char *packet );

void packet_sniffer( char *buffer );

int packet_odd( char *from, char *to, int len );

int packet_even( char *from, char *to, int len );

int packet_collect( char *odd, char *even, char *dest, int len );
```

```
/*
 * Mécanismes de configuration automatique d'une interface réseau
 * Travail d'étude 2005 - Université de Nice Sophia-Antipolis
 * AMEDRO Brian, BORDNARTCHOUK Vladimir, ROBITZER Vincent
 *
 * Prototype d'implémentation du protocole S-DHCP
 *
 * packet.c
 * Développement : AMEDRO Brian
 *
 */

#include "packet.h"
#include "sdhcp.h"

static int packet_pos;

int packet_get_option( unsigned char *packet,
                       char option,
                       int *len,
                       char *optdata)
{
  int olen, curr;

  curr = 4;
  while( packet[curr] != option ) {
    if( packet[curr] == 255 ) return -1;

    olen = packet[curr+1]*256 + packet[curr+2];
    curr += olen + 3;
  }

  *len = packet[curr+1]*256 + packet[curr+2];
  memcpy( optdata, &packet[curr+3], *len );

  optdata[*len] = 0;

  return 1;
}


int packet_init( char *packet,
                 char op,
                 char flag,
                 short xid )
{
  packet[0] = op;       // op
  packet[1] = flag;     // flag
  packet[2] = xid/256;  // xid
  packet[3] = xid%256;

  packet_pos = 4;

  return 1;
}

int packet_add( char *packet,
                char option,
                short len,
                char *data )
{
  packet[packet_pos++] = option;
  packet[packet_pos++] = len/256;
  packet[packet_pos++] = len%256;
  memcpy( &packet[packet_pos], data, len );
  packet_pos += len;

  return 1;
}

int packet_close( char *packet )
{
```

```
  packet[packet_pos++] = 255; // Marqueur de fin d'options

  return packet_pos;
}

void packet_sniffer( char *buffer ) {
  printf( "\nSNIFFER\n=======\n" );
  printf( "Header : %d / %d / %d\n",
          buffer[0], buffer[1],
          buffer[2]*256+buffer[3]%256 );

}


int packet_odd( char *from,
                char *to,
                int len )
{
  int i=0;
  int pos = 1;

  while( pos < len ) {
    to[i++] = from[pos];
    pos += 2;
  }
  to[i] = 0;

  return i;
}


int packet_even( char *from,
                 char *to,
                 int len )
{
  int i=0;
  int pos = 0;

  while( pos < len ) {
    to[i++] = from[pos];
    pos += 2;
  }
  to[i] = 0;

  return i;
}


int packet_collect( char *odd,
                    char *even,
                    char *dest,
                    int len)
{
  int i = 0;
  int pos = 0;

  while( i < len ) {
    dest[i++] = even[pos];
    if( i<len ) dest[i++] = odd[pos];
    pos++;
  }

  return i;
}
```

```
/*
 * Mécanismes de configuration automatique d'une interface réseau
 * Travail d'étude 2005 - Université de Nice Sophia-Antipolis
 * AMEDRO Brian, BORDNARTCHOUK Vladimir, ROBITZER Vincent
 *
 * Prototype d'implémentation du protocole S-DHCP
 *
 * Serveur
 * Développement : AMEDRO Brian
 *
 */


#define PORT_SERVER     17004
#define PORT_CLIENT     17005
#define BUFFER_SIZE     1024
#define RSAKEY_SIZE     1024
#define RSA_PADDING     RSA_PKCS1_OAEP_PADDING
#define SHA_SIZE        20
```

```
/*
 * Mécanismes de configuration automatique d'une interface réseau
 * Travail d'étude 2005 - Université de Nice Sophia-Antipolis
 * AMEDRO Brian, BORDNARTCHOUK Vladimir, ROBITZER Vincent
 *
 * Prototype d'implémentation du protocole S-DHCP
 *
 * Client
 * Développement : AMEDRO Brian
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <openssl/rsa.h>
#include <openssl/engine.h>
#include <openssl/sha.h>

#include "packet.h"
#include "sdhcp.h"



int main( int argc,
          char **argv )
{
  struct sockaddr_in saddr;
  struct sockaddr_in caddr;
  int s_sockfd;
  int c_sockfd;
  int slen;

  /* buffers pour les paquets imbriqués */
  char buffer_in_odd[BUFFER_SIZE];
  char buffer_in_even[BUFFER_SIZE];
  int buffer_in_sz;
  char buffer_out_odd[BUFFER_SIZE];
  char buffer_out_even[BUFFER_SIZE];
  int buffer_out_sz;

  char crypt_buf[BUFFER_SIZE];
  char buffer[BUFFER_SIZE];
  char *tmp_buffer;
  int optnum;
  int optlen;
  char optdata[BUFFER_SIZE];
  char config[BUFFER_SIZE];
  RSA *rsa_c, *rsa_s;
  int sz;
  long int sec_n, sec_m;
  char secnum[16];
  int xid = 4367;


  /* Vérification des arguments */
  if( argc != 3 ) {
    printf( "Usage : %s login password\n\n", argv[0] );
    exit(1);
  }


  /* Génération de la clé du client */
  rsa_c = RSA_generate_key( RSAKEY_SIZE, RSA_F4, NULL, NULL );
```

```
/* Socket du serveur */
bzero( &saddr, sizeof(saddr) );
saddr.sin_family = AF_INET;
saddr.sin_addr.s_addr = inet_addr( "127.0.0.1" );
saddr.sin_port = htons( PORT_SERVER );
s_sockfd = socket( AF_INET, SOCK_DGRAM, IPPROTO_UDP );

/* Socket d'écoute du client */
caddr.sin_family = AF_INET;
caddr.sin_addr.s_addr = INADDR_ANY;
caddr.sin_port = htons( PORT_CLIENT );
c_sockfd = socket( AF_INET, SOCK_DGRAM, IPPROTO_UDP );
bind( c_sockfd, (struct sockaddr *)&caddr, sizeof(caddr) );



/****
 **
 ** Début du dialogue
 **
 ****/

/* Etape 1
 *
 * Envoie : Kc
 */
printf( "Recherche du serveur SDHCP...\n" );
bzero( buffer, BUFFER_SIZE );
packet_init( buffer, 1, 0, xid );

/* Kc */
tmp_buffer = BN_bn2hex( rsa_c->n );
packet_add( buffer, 10, strlen(tmp_buffer), tmp_buffer );
tmp_buffer = BN_bn2hex( rsa_c->e );
packet_add( buffer, 11, strlen(tmp_buffer), tmp_buffer );

sz = packet_close( buffer );

sendto( s_sockfd, buffer, sz, 0,
        (struct sockaddr *)&saddr, sizeof(saddr) );



/* Etape 2
 *
 * Récéption : Ks, pair(Kc(n))
 */
sz = recvfrom( c_sockfd, buffer, BUFFER_SIZE, 0,
               (struct sockaddr *)&saddr, &slen );

rsa_s = RSA_new();

/* Ks */
packet_get_option( buffer, 10, &optlen, optdata );
BN_hex2bn( &(rsa_s->n), optdata );
packet_get_option( buffer, 11, &optlen, optdata );
BN_hex2bn( &(rsa_s->e), optdata );

/* pair(Kc(n)) */
packet_get_option( buffer, 1, &optlen, buffer_in_odd );



/* Etape 3
 *
 * Envoie : pair(Ks(m))
 */
srandom( time(NULL)*3 );
sec_m = random();

packet_init( buffer, 3, 0, xid );
```

```
sprintf( secnum, "%d", sec_m );
sz = RSA_public_encrypt( strlen(secnum),
                         secnum,
                         crypt_buf,
                         rsa_s,
                         RSA_PADDING );

/* On découpe le paquet crypté en 2 : bits pairs et impairs */
buffer_out_sz = packet_odd( crypt_buf, buffer_out_odd, sz );
packet_add( buffer, 1, buffer_out_sz, buffer_out_odd );
buffer_out_sz = packet_even( crypt_buf, buffer_out_even, sz );

sz = packet_close( buffer );

saddr.sin_port = htons( PORT_SERVER );
sendto( s_sockfd, buffer, sz, 0,
        (struct sockaddr *)&saddr, sizeof(saddr) );


/* Etape 4
 *
 * Réception : impair(Kc(n))
 */
sz = recvfrom( c_sockfd, buffer, BUFFER_SIZE, 0,
               (struct sockaddr *)&saddr, &slen );

/* impair(Kc(n)) */
packet_get_option( buffer, 1, &optlen, buffer_in_even );

/* on regroupe les bits pairs et impairs en un seul paquet */
optlen = packet_collect( buffer_in_odd, buffer_in_even,
                         optdata, 128 );

bzero( crypt_buf, BUFFER_SIZE );
sz = RSA_private_decrypt( optlen,
                          optdata,
                          crypt_buf,
                          rsa_c,
                          RSA_PADDING );

sec_n = atol( crypt_buf );



/* Etape 5
 *
 * Envoie : impair(Ks(m)), Ks(n), pair(Ks(SHA(pass,n,m)))
 */
printf( "Authentification...\n" );
packet_init( buffer, 5, 0, xid );

/* impair(Ks(m)) */
packet_add( buffer, 1, buffer_out_sz, buffer_out_even );

/* Ks(n) */
sprintf( secnum, "%d", sec_n );
sz = RSA_public_encrypt( strlen(secnum),
                         secnum,
                         crypt_buf,
                         rsa_s,
                         RSA_PADDING );
packet_add( buffer, 2, sz, crypt_buf );

/* pair(Ks(SHA(pass,n,m))) */
sprintf( optdata, "%s%d%d", argv[2], sec_n, sec_m );
tmp_buffer = SHA1( optdata, strlen(optdata), NULL );
sz = RSA_public_encrypt( strlen(tmp_buffer),
                         tmp_buffer,
                         crypt_buf,
                         rsa_s,
                         RSA_PADDING );
buffer_out_sz = packet_odd( crypt_buf, buffer_out_odd, sz );
```

```
  packet_add( buffer, 3, buffer_out_sz, buffer_out_odd );
  buffer_out_sz = packet_even( crypt_buf, buffer_out_even, sz );

  sz = packet_close( buffer );

  saddr.sin_port = htons( PORT_SERVER );
  sendto( s_sockfd, buffer, sz, 0,
          (struct sockaddr *)&saddr, sizeof(saddr) );



  /* Etape 6
   *
   * Réception : Kc(m)
   */
  sz = recvfrom( c_sockfd, buffer, BUFFER_SIZE, 0,
                 (struct sockaddr *)&saddr, &slen );

  /* Kc(m) */
  packet_get_option( buffer, 2, &optlen, optdata );
  sz = RSA_private_decrypt( optlen,
                            optdata,
                            crypt_buf,
                            rsa_c,
                            RSA_PADDING );

  if( atol(crypt_buf) != sec_m ) {
    fprintf( stderr, "/!\\ \t Laison non sûre, probablement une attaque 'passeur de seau' !!!\n"
);
    exit(1);
  }



  /* Etape 7
   *
   * Envoie Ks(login), impair(Ks(SHA(pass,n,m)))
   */
  packet_init( buffer, 7, 0, xid );

  /* impair(Ks(SHA(pass,n,m)) */
  packet_add( buffer, 3, buffer_out_sz, buffer_out_even );

  /* Ks(login) */
  sz = RSA_public_encrypt( strlen(argv[1]),
                           argv[1],
                           crypt_buf,
                           rsa_s,
                           RSA_PADDING );
  packet_add( buffer, 4, sz, crypt_buf );

  sz = packet_close( buffer );

  saddr.sin_port = htons( PORT_SERVER );
  sendto( s_sockfd, buffer, sz, 0,
          (struct sockaddr *)&saddr, sizeof(saddr) );



  /* Etape 8
   *
   * Réception : Kc(SHA(pass,n,m,config)), Kc(config)
   */
  sz = recvfrom( c_sockfd, buffer, BUFFER_SIZE, 0,
                 (struct sockaddr *)&saddr, &slen );

  /* Kc(config) */
  bzero( config, BUFFER_SIZE );
  packet_get_option( buffer, 6, &optlen, optdata );
  sz = RSA_private_decrypt( optlen,
                            optdata,
                            config,
```

```
                            rsa_c,
                            RSA_PADDING );
  printf( "Configuration : %s\n", config );

  /* Kc(SHA(pass,n,m,config)) */
  sprintf( optdata, "%s%d%d%s", argv[2], sec_n, sec_m, config );
  tmp_buffer = SHA1( optdata, strlen(optdata), NULL );
  packet_get_option( buffer, 5, &optlen, optdata );
  sz = RSA_private_decrypt( optlen,
                            optdata,
                            crypt_buf,
                            rsa_c,
                            RSA_PADDING );
  if( memcmp( tmp_buffer, crypt_buf, SHA_SIZE ) != 0 ) {
    printf( "/!\\ Serveur non autorisé ou authentification incorrecte, opération annulée.\n" );
    exit(1);
  }

  printf( "Client configuré.\n" );

  return EXIT_SUCCESS;
}
```

```c
/*
 * Mécanismes de configuration automatique d'une interface réseau
 * Travail d'étude 2005 - Université de Nice Sophia-Antipolis
 * AMEDRO Brian, BORDNARTCHOUK Vladimir, ROBITZER Vincent
 *
 * Prototype d'implémentation du protocole S-DHCP
 *
 * Serveur
 * Développement : AMEDRO Brian
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <openssl/rsa.h>
#include <openssl/engine.h>
#include <openssl/sha.h>

#include "packet.h"
#include "sdhcp.h"


#define LOGIN_MAXSIZE   50

struct client {
  int etat;
  RSA *rsa;
  int xid;
  long int n;
  long int m;
  char login[LOGIN_MAXSIZE];
  char config[BUFFER_SIZE];
};

/* Prototypes */
int get_password( char *login, char *pass );
int get_config( char *login, char *config );

int main( int argc,
          char **argv )
{
  struct sockaddr_in saddr;
  struct sockaddr_in caddr;
  int s_sockfd;
  int c_sockfd;
  int clen;
  int bufflen;
  char secnum[16];

  char buffer_in_odd[BUFFER_SIZE];
  char buffer_in_even[BUFFER_SIZE];
  int buffer_in_sz;
  char buffer_out_odd[BUFFER_SIZE];
  char buffer_out_even[BUFFER_SIZE];
  int buffer_out_sz;

  char crypt_buf[BUFFER_SIZE];
  unsigned char buffer[BUFFER_SIZE];
  char buffer2[BUFFER_SIZE];
  char *tmp_buffer;
  int sz;
  int optnum;
  int optlen;
  char optdata[BUFFER_SIZE];
```

```
RSA *rsa_s;
struct client *un_client;

printf( "Lancement du serveur...\n\n" );

/* Génération de la clé du client */
printf( "Génération d'une clé RSA de %d bits...\n", RSAKEY_SIZE );
rsa_s = RSA_generate_key( RSAKEY_SIZE, RSA_F4, NULL, NULL );


/* Initialisations */
bzero( &saddr, sizeof(saddr) );
bzero( &caddr, sizeof(caddr) );

/* Socket d'écoute du serveur */
saddr.sin_family = AF_INET;
saddr.sin_addr.s_addr = INADDR_ANY;
saddr.sin_port = htons( PORT_SERVER );
s_sockfd = socket( AF_INET, SOCK_DGRAM, IPPROTO_UDP );
bind( s_sockfd, (struct sockaddr *)&saddr, sizeof(saddr) );

/* Socket des clients */
c_sockfd = socket( AF_INET, SOCK_DGRAM, IPPROTO_UDP );


/* Boucle principale */
while( 1 ) {
  printf( "\n\n\nAttente d'un client...\n" );
  bzero( buffer, BUFFER_SIZE );
  sz = recvfrom( s_sockfd, buffer, BUFFER_SIZE, 0,
                 (struct sockaddr *)&caddr, &clen );

  /* Etape 1
   *
   * Réception :  Kc
   */
  printf( "Etape 1 : Réception de Kc\n" );
  un_client = malloc(sizeof(struct client));
  un_client->rsa = RSA_new();
  un_client->xid = 4367;

  /* Kc */
  packet_get_option( buffer, 10, &optlen, optdata );
  BN_hex2bn( &(un_client->rsa->n), optdata );
  packet_get_option( buffer, 11, &optlen, optdata );
  BN_hex2bn( &(un_client->rsa->e), optdata );



  /* Etape 2
   *
   * Envoie : Ks, pair(Kc(n))
   */
  printf( "Etape 2 : Envoie de Ks, pair(Kc(n))\n" );
  srandom( time(NULL) );
  un_client->n = random();

  bzero( buffer, BUFFER_SIZE );
  packet_init( buffer, 2, 0, un_client->xid );

  /* Ks */
  tmp_buffer = BN_bn2hex( rsa_s->n );
  packet_add( buffer, 10, strlen(tmp_buffer), tmp_buffer );
  tmp_buffer = BN_bn2hex( rsa_s->e );
  packet_add( buffer, 11, strlen(tmp_buffer), tmp_buffer );

  /* pair(Kc(n)) */
  sprintf( secnum, "%d", un_client->n );
  sz = RSA_public_encrypt( strlen(secnum)+1,
                           secnum,
                           crypt_buf,
                           un_client->rsa,
```

```
                          RSA_PADDING );
buffer_out_sz = packet_odd( crypt_buf, buffer_out_odd, sz );
packet_add( buffer, 1, buffer_out_sz, buffer_out_odd );
buffer_out_sz = packet_even( crypt_buf, buffer_out_even, sz );

sz = packet_close( buffer );

caddr.sin_port = htons( PORT_CLIENT );
sendto( c_sockfd, buffer, sz, 0,
        (struct sockaddr *)&caddr, clen );


/* Etape 3
 *
 * Réception : pair(Ks(m))
 */
printf( "Etape 3 : Réception de pair(Ks(m))\n" );
bzero( buffer, BUFFER_SIZE );
sz = recvfrom( s_sockfd, buffer, BUFFER_SIZE, 0,
               (struct sockaddr *)&caddr, &clen );

/* pair(Ks(m)) */
packet_get_option( buffer, 1, &optlen, buffer_in_odd );



/* Etape 4
 *
 * Envoie : impair(Kc(n))
 */
printf( "Etape 4 : Envoie de impair(Kc(n))\n" );
bzero( buffer, BUFFER_SIZE );
packet_init( buffer, 4, 0, un_client->xid );

/* impair(Kc(n)) */
packet_add( buffer, 1, buffer_out_sz, buffer_out_even );

sz = packet_close( buffer );
caddr.sin_port = htons( PORT_CLIENT );
sendto( c_sockfd, buffer, sz, 0,
        (struct sockaddr *)&caddr, clen );



/* Etape 5
 *
 * Réception : impair(Ks(m)), Ks(n), pair(Ks(SHA(pass,n,m)))
 */
printf( "Etape 5 : Réception de impair(Ks(m)), Ks(n), pair(Ks(SHA(pass,n,m)))\n" );
bzero( buffer, BUFFER_SIZE );
sz = recvfrom( s_sockfd, buffer, BUFFER_SIZE, 0,
               (struct sockaddr *)&caddr, &clen );

/* Ks(n) */
packet_get_option( buffer, 2, &optlen, optdata );
bzero( crypt_buf, BUFFER_SIZE );
sz = RSA_private_decrypt( optlen,
                          optdata,
                          crypt_buf,
                          rsa_s,
                          RSA_PADDING );
if( atol(crypt_buf) != un_client->n ) {
  printf( "/!\\ \t Laison écoutée !!!\n" );
  continue;
}

/*  impair(Ks(m)) */
packet_get_option( buffer, 1, &optlen, buffer_in_even );
optlen = packet_collect( buffer_in_odd, buffer_in_even,
                         optdata, 128 );
bzero( crypt_buf, BUFFER_SIZE );
```

```
sz = RSA_private_decrypt( optlen,
                          optdata,
                          crypt_buf,
                          rsa_s,
                          RSA_PADDING );
un_client->m = atol(crypt_buf);

/* pair(Ks(SHA(pass,n,m))) */
packet_get_option( buffer, 3, &optlen, buffer_in_odd );
buffer_in_odd[optlen] = 0;



/* Etape 6
 *
 * Envoie : Kc(m)
 */
printf( "Etape 6 : Envoie de Kc(m)\n" );
bzero( buffer, BUFFER_SIZE );
packet_init( buffer, 6, 0, un_client->xid );

/* Kc(m) */
sprintf( secnum, "%d", un_client->m );
sz = RSA_public_encrypt( strlen(secnum),
                         secnum,
                         crypt_buf,
                         un_client->rsa,
                         RSA_PADDING );
packet_add( buffer, 2, sz, crypt_buf );

sz = packet_close( buffer );
caddr.sin_port = htons( PORT_CLIENT );
sendto( c_sockfd, buffer, sz, 0,
        (struct sockaddr *)&caddr, clen );



/* Etape 7
 *
 * Récéption : Ks(login), impair(Ks(SHA(pass,n,m)))
 */
printf( "Etape 7 : Récéption de Ks(login), impair(Ks(SHA(pass,n,m)))\n" );
bzero( buffer, BUFFER_SIZE );
sz = recvfrom( s_sockfd, buffer, BUFFER_SIZE, 0,
               (struct sockaddr *)&caddr, &clen );

/* Ks(login) */
bzero( un_client->login, LOGIN_MAXSIZE );
packet_get_option( buffer, 4, &optlen, optdata );
sz = RSA_private_decrypt( optlen,
                          optdata,
                          un_client->login,
                          rsa_s,
                          RSA_PADDING );

/* impair(Ks(SHA(pass,n,m)) */
packet_get_option( buffer, 3, &optlen, buffer_in_even );
optlen = packet_collect( buffer_in_odd, buffer_in_even,
                         optdata, 128 );
bzero( crypt_buf, BUFFER_SIZE );
sz = RSA_private_decrypt( optlen,
                          optdata,
                          crypt_buf,
                          rsa_s,
                          RSA_PADDING );
get_password( un_client->login, optdata );
sprintf( buffer2, "%s%d%d", optdata, un_client->n, un_client->m );
tmp_buffer = SHA1( buffer2, strlen(buffer2), NULL );
if( memcmp( crypt_buf, tmp_buffer, SHA_SIZE ) != 0 ) {
  printf( "Authentification du client incorrecte.\n" );
  continue;
} else {
```

```
      printf( "Client authentifié, envoie de la configuration\n" );
    }


    /* Etape 8
     *
     * Envoie : Kc(SHA(pass,n,m,config)), Kc(config)
     */
    printf( "Etape 8 : Envoie de Kc(SHA(pass,n,m,config)), Kc(config)\n" );
    bzero( buffer, BUFFER_SIZE );
    packet_init( buffer, 8, 0, 4367 );

    /* Kc(SHA(pass,n,m,config)) */
    get_config( un_client->login, un_client->config );
    sprintf( buffer2, "%s%d%d%s", optdata, un_client->n, un_client->m, un_client->config );
    tmp_buffer = SHA1( buffer2, strlen(buffer2), NULL );
    sz = RSA_public_encrypt( strlen(tmp_buffer),
                             tmp_buffer,
                             crypt_buf,
                             un_client->rsa,
                             RSA_PADDING );
    packet_add( buffer, 5, sz, crypt_buf );

    /* Kc(config) */
    sz = RSA_public_encrypt( strlen(un_client->config),
                             un_client->config,
                             crypt_buf,
                             un_client->rsa,
                             RSA_PADDING );
    packet_add( buffer, 6, sz, crypt_buf );


    sz = packet_close( buffer );
    caddr.sin_port = htons( PORT_CLIENT );
    sendto( c_sockfd, buffer, sz, 0,
            (struct sockaddr *)&caddr, clen );



  }

return EXIT_SUCCESS;
}


/* Deux fonctions qui simulent un accès à la base de données */
int get_password( char *login,
  char *pass )
{
strcpy( pass, "toto" );
return 1;
}
int get_config( char *login,
  char *config )
{
  strcpy( config, "IP:192.168.1.1 MASQUE:255.255.255.0" );

  return 1;
}
```