CODIGO FUENTE DESARROLLADO DURANTE EL PFC PARA LA IMPLEMENTACION DEL MAPEADO DE DISCOS LOCALES. GESTIÓN Y SINCRONIZACIÓN DEL PORTAPAPELES.

AUTOR : JAVIER CAVERNI IBÁÑEZ

PONENTE: ÁLVARO ALESANCO


Departamento de Ingeniería Electrónica y Comunicaciones

INTRODUCCIÓN

En este anexo se incluye únicamente el código fuente modificado por el alumno para la consecución de los objetivos establecidos en este Proyecto Fin de Carrera.

Cómo ya se ha explicado en la memoria, dichas modificaciones fueron realizadas sobre una base de código fuente ya existente. Dicho código de partida se encuentra en la versión 0.4.1 del proyecto XRDP que era la última versión estable de dicho proyecto en el momento del comienzo de las prácticas.

La organización de este anexo nos va a permitir conocer en detalle cada una de las funciones que fueron modificadas así como su ubicación exacta dentro de la estructura del proyecto XRDP.

Así pues, cada sección tendrá como título el repertorio donde se incluye la función modificada así como el código modificado de la misma, con el objetivo de presentar a los miembros del jurado y de la comunidad "Open Source" las modificaciones llevadas a cabo.

Los módulos modificados han sido los siguientes :

- Xrdp/common/os_calls.c
- Xrdp/libxrdp/xrdp_channel.c
- Xrdp/rdp/rdp.c
- Xrdp/rdp/rdp_iso.c
- Xrdp/rdp/rdp_mcs.c
- Xrdp/rdp/rdp_rdp.c
- Xrdp/rdp/rdp_sec.c

Xrdp/common/oscalls.c

```
/*****************************************************************************/
/* Function created in order to print messages in /var/log/xrdp_log.log file*/
int fonc_log(char* text)
{
   FILE *fd_debug;
   fd_debug = fopen("/var/log/xrdp_log.log","a");
   if (fd_debug==NULL)
   {
       printf("Error opening the file");
   }
   fprintf(fd_debug,"\n/************************/\n");
   fprintf(fd_debug,text);
   fprintf(fd_debug,"\n");
   fclose(fd_debug);
   return 0;
 }
```

Xrdp/libxrdp/xrdp_channel.c

```
/*****************************************************************************/
/* returns error */
/* This sends data out to the secure layer. */
int APP_CC
xrdp_channel_send(struct xrdp_channel* self, struct stream* s, int channel_id,
                  int total_data_len, int flags)
{
  int chanid;
  struct mcs_channel_item* channel;
  chanid = (channel_id - MCS_GLOBAL_CHANNEL) - 1;
  channel = xrdp_channel_get_item(self, chanid);
  if (channel == 0)
  {
    fonc_log(" channel == 0");
    return 1;
  }
  s_pop_layer(s, channel_hdr);
  out_uint32_le(s, total_data_len);
  if (channel->flags & CHANNEL_OPTION_SHOW_PROTOCOL)
  {
    flags |= CHANNEL_FLAG_SHOW_PROTOCOL;
  }
  out_uint32_le(s, flags);
  if (xrdp_sec_send(self->sec_layer, s, channel->chanid) != 0)
  {
    fonc_log(" xrdp_sec_send failed");
    return 1;
  }
  return 0;
}
```

Xrdp/rdp/rdp.c

```
/*****************************************************************************/
/* return error */
int DEFAULT_CC
lib_mod_connect(struct mod* mod, struct list* channel_list_param, struct client_info* client_info)
{
  fonc_log("in rdp mod connect, connecting to RDP server");
  DEBUG(("in lib_mod_connect"));
  mod->client_info = client_info;
  mod->channel_list = channel_list_param;

  /* clear screen */
  mod->server_begin_update(mod);
  mod->server_set_fgcolor(mod, 0);
  mod->server_fill_rect(mod, 0, 0, mod->width, mod->height);
  mod->server_end_update(mod);
  /* connect */
  if (rdp_rdp_connect(mod->rdp_layer, mod->ip, mod->port, mod->channel_list) == 0)
  {
    mod->sck = mod->rdp_layer->sec_layer->mcs_layer->iso_layer->tcp_layer->sck;
    g_tcp_set_non_blocking(mod->sck);
    g_tcp_set_no_delay(mod->sck);
    mod->sck_obj = g_create_wait_obj_from_socket(mod->sck, 0);
    DEBUG(("out lib_mod_connect"));
    fonc_log("out rdp mod connect, succesfully connected to RDP server");
    return 0;
  }
  DEBUG(("out lib_mod_connect error"));
  return 1;
}
/*****************************************************************************/
/* return error */
int DEFAULT_CC
lib_mod_event(struct mod* mod, int msg, long param1, long param2,
              long param3, long param4)
{
  struct stream* s;

  if (!mod->up_and_running)
  {
    return 0;
  }
  DEBUG(("in lib_mod_event"));
  make_stream(s);
  init_stream(s, 8192 * 2);
  switch (msg)
  {
    case 15:
      rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_SCANCODE,
                         param4, param3, 0);
      break;
    case 16:
      rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_SCANCODE,
                         param4, param3, 0);
      break;
    case 17:
      rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_SYNCHRONIZE,
                         param4, param3, 0);
      break;
```

```
case 100:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_MOVE, param1, param2);
  break;
case 101:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_BUTTON1, param1, param2);
  break;
case 102:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_BUTTON1 | MOUSE_FLAG_DOWN,
                param1, param2);
  break;
case 103:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_BUTTON2, param1, param2);
  break;
case 104:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_BUTTON2 | MOUSE_FLAG_DOWN,
                param1, param2);
  break;
case 105:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_BUTTON3, param1, param2);
  break;
case 106:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_BUTTON3 | MOUSE_FLAG_DOWN,
                param1, param2);
  break;
case 107:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_BUTTON4, param1, param2);
  break;
case 108:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_BUTTON4 | MOUSE_FLAG_DOWN,
                param1, param2);
  break;
case 109:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_BUTTON5, param1, param2);
  break;
case 110:
  rdp_rdp_send_input(mod->rdp_layer, s, 0, RDP_INPUT_MOUSE,
                MOUSE_FLAG_BUTTON5 | MOUSE_FLAG_DOWN,
                param1, param2);
  break;
case 200:
  rdp_rdp_send_invalidate(mod->rdp_layer, s,
              (param1 >> 16) & 0xffff, param1 & 0xffff,
              (param2 >> 16) & 0xffff, param2 & 0xffff);
  break;
case 0x5555:
  fonc_log("In rdp_rdp_send_redirect_pdu case ---> processing redirection packet");
  rdp_rdp_send_redirect_pdu(mod->rdp_layer, param1, param2, param3, param4);
  break;
default:
  fonc_log("In rdp.c --> default case in lib_mod_event");
```

```
      break;
  }
  free_stream(s);
  DEBUG(("out lib_mod_event"));

  return 0;
}
/*****************************************************************************/
/* return error */
int DEFAULT_CC
lib_mod_end(struct mod* mod)
{
  rdp_rdp_delete(mod->rdp_layer);
  mod->rdp_layer = 0;
  free_stream(mod->in_s);
  mod->in_s = 0;
  if (mod->sck_obj != 0)
  {
    g_delete_wait_obj_from_socket(mod->sck_obj);
    mod->sck_obj = 0;
  }
  if (mod->sck != 0)
  {
    g_tcp_close(mod->sck);
    mod->sck = 0;
  }
  list_delete(mod->channel_list);
  return 0;
}
/*****************************************************************************/
struct mod* EXPORT_CC
mod_init(void)
{
  struct mod* mod;

  DEBUG(("in mod_init"));
  mod = (struct mod*)g_malloc(sizeof(struct mod), 1);
  mod->size = sizeof(struct mod);
  mod->handle = (long)mod;
  mod->mod_connect = lib_mod_connect;
  mod->mod_start = lib_mod_start;
  mod->mod_event = lib_mod_event;
  mod->mod_signal = lib_mod_signal;
  mod->mod_end = lib_mod_end;
  mod->mod_set_param = lib_mod_set_param;
  mod->mod_get_wait_objs = lib_mod_get_wait_objs;
  mod->mod_check_wait_objs = lib_mod_check_wait_objs;
  mod->rdp_layer = rdp_rdp_create(mod);
  mod->channel_list = list_create();
  DEBUG(("out mod_init"));
  return mod;
}
```

Xrdp/rdp/rdp_iso.c

```
/*****************************************************************************/
/* returns error */
static int APP_CC
rdp_iso_recv_msg(struct rdp_iso* self, struct stream* s, int* code)
{
  int ver;
  int len;

  *code = 0;
  if (rdp_tcp_recv(self->tcp_layer, s, 4) != 0)
  {
    fonc_log("out rdp_iso_recv_msg error rdp_tcp_recv 1 failed");
    DEBUG((" out rdp_iso_recv_msg error rdp_tcp_recv 1 failed"));
    return 1;
  }
  in_uint8(s, ver);
  self->mcs_layer->sec_layer->rdp_layer->version =  ver;

  if (ver == 3)
  {
    in_uint8s(s, 1);
    in_uint16_be(s, len);
  }
  else
  {
    in_uint8(s, len);
    if (len & 0x80)
    {
      in_uint8(s, len);
    }
  }
  /* We must verify the length of incoming header */
  if (len < 4)
  {
    DEBUG((" out rdp_iso_recv_msg error, bad error header"));
    fonc_log("out rdp_iso_recv_msg error, bad error header");
    return 1;
  }
  if (rdp_tcp_recv(self->tcp_layer, s, len - 4) != 0)
  {
    fonc_log(" rdp_tcp_recv 2 failed");
    DEBUG((" out rdp_iso_recv_msg error rdp_tcp_recv 2 failed"));
    return 1;
  }
  if (self->mcs_layer->sec_layer->rdp_layer->version != 3)
  {
    return 0;
  }
  in_uint8s(s, 1);
  in_uint8(s, *code);
  if (*code == ISO_PDU_DT)
  {
    in_uint8s(s, 1);
  }
  else
  {
    in_uint8s(s, 5);
  }
```

```
  return 0;
}



/*****************************************************************************/
static int APP_CC
rdp_iso_send_msg(struct rdp_iso* self, struct stream* s, int code)
{
  int length;

  length = 30 + g_strlen(self->mcs_layer->sec_layer->rdp_layer->mod->username);

  if (rdp_tcp_init(self->tcp_layer, s) != 0)
  {
    return 1;
  }
  if (code == ISO_PDU_CR)
  {
    out_uint8(s, 3);  /* version */
    out_uint8(s, 0);  /* reserved */
    out_uint16_be(s, length); /* length */
    out_uint8(s, length - 5);  /* hdrlen */
    out_uint8(s, ISO_PDU_CR);
    out_uint16_le(s, 0); /* dest ref*/
    out_uint16_le(s, 0); /* src ref*/
    out_uint8(s, 0); /* class */

            out_uint8p(s, "Cookie: mstshash=", g_strlen("Cookie: mstshash="));
            out_uint8p(s, self->mcs_layer->sec_layer->rdp_layer->mod->username,
              length - 30);

            out_uint8(s, 0x0d);         /* Unknown */
            out_uint8(s, 0x0a);         /* Unknown */
  }
  else
  {
    out_uint8(s, 3);  /* version */
    out_uint8(s, 0);  /* reserved */
    out_uint16_be(s, 11); /* length */
    out_uint8(s, 6);  /* hdrlen */
    out_uint8(s, code);
    out_uint16_le(s, 0); /* dest ref*/
    out_uint16_le(s, 0); /* src ref*/
    out_uint8(s, 0); /* class */
  }

  //printf("in rdp_iso_send, sending data from s->p \n");
  //g_hexdump(s->p,(s->end - s->p));
  s_mark_end(s);
  if (rdp_tcp_send(self->tcp_layer, s) != 0)
  {
    fonc_log("in rdp_iso_send_message, rdp_tcp_send failed");
    return 1;
  }
  return 0;
}
```

Xrdp/rdp/rdp_mcs.c

```
/*****************************************************************************/
struct rdp_mcs* APP_CC
rdp_mcs_create(struct rdp_sec* owner,
               struct stream* client_mcs_data,
               struct stream* server_mcs_data)
{
  struct rdp_mcs* self;

  self = (struct rdp_mcs*)g_malloc(sizeof(struct rdp_mcs), 1);
  self->sec_layer = owner;
  self->userid = 1;
  self->client_mcs_data = client_mcs_data;
  self->server_mcs_data = server_mcs_data;
  self->iso_layer = rdp_iso_create(self);
  self->channel_list = list_create();
  return self;
}

/*****************************************************************************/

/* returns error */
int APP_CC
rdp_mcs_recv(struct rdp_mcs* self, struct stream* s, int* chan)
{
  int appid;
  int opcode;
  int len;

  DEBUG((" in rdp_mcs_recv"));
  if (rdp_iso_recv(self->iso_layer, s) != 0)
  {
    fonc_log("error in rdp_iso_recv");
    return 1;
  }
  if (self->sec_layer->rdp_layer->version != 3)
  {
    return 0;
  }
  in_uint8(s, opcode);
  appid = opcode >> 2;
  if (appid != MCS_SDIN)
  {
    DEBUG((" out rdp_mcs_recv error"));
    return 1;
  }
  in_uint8s(s, 2);
  in_uint16_be(s, *chan);
  in_uint8s(s, 1);
  in_uint8(s, len);
  if (len & 0x80)
  {
    in_uint8s(s, 1);
  }
  DEBUG((" out rdp_mcs_recv"));
  return 0;
}

/*****************************************************************************/
```

```
/* returns error */
static int APP_CC
rdp_mcs_send_connection_initial(struct rdp_mcs* self)
{
  int data_len;
  int len;
  struct stream* s;

  make_stream(s);
  init_stream(s, 8192);
  data_len = self->client_mcs_data->end - self->client_mcs_data->data;
  len = 7 + 3 * 34 + 4 + data_len;
  if (rdp_iso_init(self->iso_layer, s) != 0)
  {
    free_stream(s);
    return 1;
  }
  rdp_mcs_ber_out_header(self, s, MCS_CONNECT_INITIAL, len);
  rdp_mcs_ber_out_header(self, s, BER_TAG_OCTET_STRING, 0); /* calling domain */
  rdp_mcs_ber_out_header(self, s, BER_TAG_OCTET_STRING, 0); /* called domain */
  rdp_mcs_ber_out_header(self, s, BER_TAG_BOOLEAN, 1);
  out_uint8(s, 0xff); /* upward flag */
  rdp_mcs_out_domain_params(self, s, 34, 2, 0, 0xffff); /* target params */
  rdp_mcs_out_domain_params(self, s, 1, 1, 1, 0x420); /* min params */
  rdp_mcs_out_domain_params(self, s, 0xffff, 0xfc17, 0xffff, 0xffff); /* max params */
  rdp_mcs_ber_out_header(self, s, BER_TAG_OCTET_STRING, data_len);
  out_uint8p(s, self->client_mcs_data->data, data_len);
  s_mark_end(s);

  DEBUG((" in rdp_mcs_send_connect_initial"));
  if (rdp_iso_send(self->iso_layer, s) != 0)
  {
    free_stream(s);
    return 1;
  }
  free_stream(s);
  return 0;
}

/*****************************************************************************/
/* returns error */
int APP_CC
rdp_mcs_connect(struct rdp_mcs* self, char* ip, char* port, struct list* channel_list_param)
{
  int index;
  int num_channels;
  struct mcs_channel_item* channel_item;
  char text[256];
  DEBUG((" in rdp_mcs_connect"));
  if (rdp_iso_connect(self->iso_layer, ip, port) != 0)
  {
    DEBUG((" out rdp_mcs_connect error rdp_iso_connect failed"));
    return 1;
  }
  rdp_mcs_send_connection_initial(self);
  if (rdp_mcs_recv_connection_response(self) != 0)
  {
    rdp_iso_disconnect(self->iso_layer);
    DEBUG((" out rdp_mcs_connect error rdp_mcs_recv_connection_response \
failed"));
```

```
    return 1;
  }
  rdp_mcs_send_edrq(self);
  rdp_mcs_send_aurq(self);
  if (rdp_mcs_recv_aucf(self) != 0)
  {
    rdp_iso_disconnect(self->iso_layer);
    DEBUG((" out rdp_mcs_connect error rdp_mcs_recv_aucf failed"));
    return 1;
  }
  rdp_mcs_send_cjrq(self, self->userid + 1001);
  if (rdp_mcs_recv_cjcf(self) != 0)
  {
    rdp_iso_disconnect(self->iso_layer);
    DEBUG((" out rdp_mcs_connect error rdp_mcs_recv_cjcf 1 failed"));
    return 1;
  }
  rdp_mcs_send_cjrq(self, MCS_GLOBAL_CHANNEL);
  if (rdp_mcs_recv_cjcf(self) != 0)
  {
    rdp_iso_disconnect(self->iso_layer);
    DEBUG((" out rdp_mcs_connect error rdp_mcs_recv_cjcf 2 failed"));
    return 1;
  }
  num_channels = self->channel_list->count;
  for (index = 0; index < num_channels; index++)
  {
    channel_item = (struct mcs_channel_item*)list_get_item(self->channel_list, index);
    rdp_mcs_send_cjrq(self, channel_item->chanid);
    fonc_log(" In rdp_mcs_connect, channel_id confirmed: ");
    g_snprintf(text, 255, "%d", channel_item->chanid);
    fonc_log(text);
    if (rdp_mcs_recv_cjcf(self) != 0)
    {
      fonc_log("out rdp_mcs_connect error, rdp_mcs_recv_cjcf failed--> exiting");
      rdp_iso_disconnect(self->iso_layer);
      DEBUG((" out rdp_mcs_connect error rdp_mcs_recv_cjcf failed"));
      return 1;
    }
  }
  DEBUG((" out rdp_mcs_connect"));
  return 0;
}


/*****************************************************************************/
/* returns error */
int APP_CC
rdp_mcs_send(struct rdp_mcs* self, struct stream* s, int chan_id)
{
  int len;
  char text[256];

  s_pop_layer(s, mcs_hdr);
  len = (s->end - s->p) - 8;
  len = len | 0x8000;
  out_uint8(s, MCS_SDRQ << 2);
  out_uint16_be(s, self->userid);
  out_uint16_be(s, chan_id);
  out_uint8(s, 0x70);
```

```
  out_uint16_be(s, len);
  if (chan_id != MCS_GLOBAL_CHANNEL)
  {
    if (len > CHANNEL_CHUNK_LENGTH)
    {
      fonc_log(" Out of rdp_mcs_send, length exceeded: ");
      g_snprintf(text, 255, "%d", len);
      fonc_log(text);
    }
  }
  if (rdp_iso_send(self->iso_layer, s) != 0)
  {
    fonc_log("out rdp_mcs_send, rdp_iso_send failed");
    return 1;
  }
  return 0;
}
```

Xrdp/rdp/rdp_rdp.c

```
/*****************************************************************************/
struct rdp_rdp* APP_CC
rdp_rdp_create(struct mod* owner)
{
  struct rdp_rdp* self;

  self = (struct rdp_rdp*)g_malloc(sizeof(struct rdp_rdp), 1);
  self->mod = owner;
  self->sec_layer = rdp_sec_create(self);
  self->bitmap_compression = 1;
  self->bitmap_cache = 1;
  self->desktop_save = 0;
  self->orders = rdp_orders_create(self);
  self->rec_mode = 0;
  self->use_rdp5 = 0;
  self->version = 0;
  return self;
}

/*****************************************************************************/
/* Send an RDP packet */
int APP_CC
rdp_rdp_send(struct rdp_rdp* self, struct stream* s, int pdu_type)
{
  int len;
  int sec_flags;

  s_pop_layer(s, rdp_hdr);
  len = s->end - s->p;
  out_uint16_le(s, len);

  /* Added in order to adapt to version 5 packet */
  if (self->use_rdp5)
  {
    out_uint16_le(s, RDP_PDU_DATA | 0x10);
    out_uint16_le(s, self->sec_layer->mcs_layer->userid);
    out_uint32_le(s, self->share_id);
    out_uint8(s, 0);  /* pad */
    out_uint8(s, 1);  /* stream id*/
    out_uint16_le(s, (len - 14));
    out_uint8(s, pdu_type);
    out_uint8(s, 0);  /* compress type */
    out_uint16_le(s, 0);  /* compress length */
  }
  else
  {
    out_uint16_le(s, pdu_type | 0x10);
    out_uint16_le(s, self->sec_layer->mcs_layer->userid);
  }
  sec_flags = SEC_ENCRYPT;
  if (rdp_sec_send(self->sec_layer, s, sec_flags, MCS_GLOBAL_CHANNEL) != 0)
  {
    return 1;
  }
  return 0;
}
```

```
/*****************************************************************************/
/* Send an RDP data packet */
int APP_CC
rdp_rdp_send_data(struct rdp_rdp* self, struct stream* s, int pdu_data_type, int chan_id)
{
  int len;
  int sec_flags;

  s_pop_layer(s, rdp_hdr);
  len = s->end - s->p;
  out_uint16_le(s, len);
  out_uint16_le(s, RDP_PDU_DATA | 0x10);
  out_uint16_le(s, self->sec_layer->mcs_layer->userid);
  out_uint32_le(s, self->share_id);
  out_uint8(s, 0); /*pad*/
  out_uint8(s, 1); /*stream id*/
  out_uint16_le(s, len - 14);
  out_uint8(s, pdu_data_type);
  out_uint8(s, 0); /* compress type */
  out_uint16_le(s, 0); /* compress len */
  sec_flags = SEC_ENCRYPT;

  if (rdp_sec_send(self->sec_layer, s, sec_flags, chan_id) != 0)
  {
    fonc_log("Error sending in rdp_sec_send");
    return 1;
  }
  return 0;
}

/*****************************************************************************/
/* Send a confirm active PDU */
static int APP_CC
rdp_rdp_send_confirm_active(struct rdp_rdp* self, struct stream* s)
{
  int sec_flags;
  int caplen;

  sec_flags = SEC_ENCRYPT;
  //sec_flags = RDP5_FLAG | SEC_ENCRYPT;
  caplen = RDP_CAPLEN_GENERAL + RDP_CAPLEN_BITMAP + RDP_CAPLEN_ORDER +
           RDP_CAPLEN_BMPCACHE + RDP_CAPLEN_COLCACHE +
           RDP_CAPLEN_ACTIVATE + RDP_CAPLEN_CONTROL +
           RDP_CAPLEN_POINTER_MONO + RDP_CAPLEN_SHARE +
           0x58 + 0x08 + 0x08 + 0x34 /* unknown caps */  +
           4 /* w2k fix, why? */ ;
  if (rdp_sec_init(self->sec_layer, s, sec_flags) != 0)
  {
    return 1;
  }
  out_uint16_le(s, 2 + 14 + caplen + sizeof(RDP_SOURCE));
  out_uint16_le(s, (RDP_PDU_CONFIRM_ACTIVE | 0x10)); /* Version 1 */
  out_uint16_le(s, (self->sec_layer->mcs_layer->userid + 1001));
  out_uint32_le(s, self->share_id);
  out_uint16_le(s, 0x3ea); /* userid */
  out_uint16_le(s, sizeof(RDP_SOURCE));
  out_uint16_le(s, caplen);
  out_uint8p(s, RDP_SOURCE, sizeof(RDP_SOURCE));
  out_uint16_le(s, 0xd); /* num_caps */
```

```
  out_uint8s(s, 2); /* pad */
  rdp_rdp_out_general_caps(self, s);
  rdp_rdp_out_bitmap_caps(self, s);
  rdp_rdp_out_order_caps(self, s);

  if(self->use_rdp5 == 0)
  {
    rdp_rdp_out_bmpcache_caps(self, s);
  }
  else
  {
    rdp_rdp_out_bmpcache2_caps(self, s);
  }

  rdp_rdp_out_colcache_caps(self, s);
  rdp_rdp_out_activate_caps(self, s);
  rdp_rdp_out_control_caps(self, s);
  rdp_rdp_out_pointer_caps(self, s);
  rdp_rdp_out_share_caps(self, s);
  rdp_rdp_out_unknown_caps(self, s, 0x0d, 0x58, caps_0x0d); /* international? */
  rdp_rdp_out_unknown_caps(self, s, 0x0c, 0x08, caps_0x0c);
  rdp_rdp_out_unknown_caps(self, s, 0x0e, 0x08, caps_0x0e);
  rdp_rdp_out_unknown_caps(self, s, 0x10, 0x34, caps_0x10); /* glyph cache? */
  s_mark_end(s);
  if (rdp_sec_send(self->sec_layer, s, sec_flags, MCS_GLOBAL_CHANNEL) != 0)
  {
    return 1;
  }
  return 0;
}



/*****************************************************************************/
int APP_CC
rdp_rdp_send_login_info(struct rdp_rdp* self, int flags)
{
  int len_domain;
  int len_username;
  int len_password;
  int len_program;
  int len_directory;
  int len_ip_address;
  int len_dll;
  int sec_flags;
  int length_stream;
  struct stream* s;
  struct hostent* hent;
  time_t t = time(NULL);
  time_t tzone;
  struct in_addr **adr;
  char* ip_source;

  hent = NULL;
  hent = (struct hostent*)g_malloc(sizeof(struct hostent), 1);
  self->mod->ip_source = NULL;
  self->mod->ip_source = g_malloc(256, 1);

  DEBUG(("in rdp_rdp_send_login_info"));
  fonc_log("in rdp_rdp_send_login_info");
```

```
make_stream(s);
init_stream(s, 8192);

fonc_log(" after init_stream");

hent = gethostbyname(self->mod->hostname);
//adr = (struct in_addr **)hent->h_addr;
//self->mod->ip_source = inet_ntoa(**adr);

len_domain = 2 * g_strlen(self->mod->domain);
len_username = 2 * g_strlen(self->mod->username);
len_password = 2 * g_strlen(self->mod->password);
len_program = 2 * g_strlen(self->mod->program);
len_directory = 2 * g_strlen(self->mod->directory);
len_ip_address = 2 * g_strlen(self->mod->ip_source);
len_dll = 2 * g_strlen("C:\\WINNT\\System32\\mstscax.dll");


sec_flags = SEC_LOGON_INFO | SEC_ENCRYPT;


if (rdp_sec_init(self->sec_layer, s, sec_flags) != 0)
{
    free_stream(s);
    DEBUG(("out rdp_rdp_send_login_info error 1"));
    return 1;
}

if(!self->use_rdp5)
{
  out_uint32_le(s, 0);
  out_uint32_le(s, flags);
  out_uint16_le(s, len_domain);
  out_uint16_le(s, len_username);
  out_uint16_le(s, len_password);
  out_uint16_le(s, len_program);
  out_uint16_le(s, len_directory);
  rdp_rdp_out_unistr(s, self->mod->domain);
  rdp_rdp_out_unistr(s, self->mod->username);
  rdp_rdp_out_unistr(s, self->mod->password);
  rdp_rdp_out_unistr(s, self->mod->program);
  rdp_rdp_out_unistr(s, self->mod->directory);
}
else
{
  flags |= RDP_LOGON_BLOB;
  out_uint32_le(s, 0);
  out_uint32_le(s, flags);
  out_uint16_le(s, len_domain);
  out_uint16_le(s, len_username);
  if (flags & RDP_LOGON_AUTO)
  {
    out_uint16_le(s, len_password);
  }
  if (flags & RDP_LOGON_BLOB && ! (flags & RDP_LOGON_AUTO))
  {
    out_uint16_le(s, 0);
  }
  out_uint16_le(s, len_program);
  out_uint16_le(s, len_directory);
```

```
if ( 0 < len_domain)
{
  rdp_rdp_out_unistr(s, self->mod->domain);
}
else
{
  out_uint16_le(s, 0);
}
rdp_rdp_out_unistr(s, self->mod->username);
if (flags & RDP_LOGON_AUTO)
{
  rdp_rdp_out_unistr(s, self->mod->password);
}
else
{
  out_uint16_le(s, 0);
}
if ( 0 < len_program)
{
  rdp_rdp_out_unistr(s, self->mod->program);
}
else
{
  out_uint16_le(s, 0);
}
if (len_directory < 0)
{
  rdp_rdp_out_unistr(s, self->mod->directory);
}
else
{
  out_uint16_le(s, 0);
}
out_uint16_le(s, 2);
out_uint16_le(s, len_ip_address + 2);
rdp_rdp_out_unistr(s, self->mod->ip_source);
out_uint16_le(s, len_dll + 2);
rdp_rdp_out_unistr(s, "C:\\WINNT\\System32\\mstscax.dll");


tzone = (mktime(gmtime(&t)) - mktime(localtime(&t))) / 60;
out_uint32_le(s, tzone);

rdp_rdp_out_unistr(s, "GTB, normaltid");
out_uint8s(s, 62 - 2 * g_strlen("GTB, normaltid"));

out_uint32_le(s, 0x0a0000);
out_uint32_le(s, 0x050000);
out_uint32_le(s, 3);
out_uint32_le(s, 0);
out_uint32_le(s, 0);

rdp_rdp_out_unistr(s, "GTB, sommartid");
out_uint8s(s, 62 - 2 * g_strlen("GTB, sommartid"));

out_uint32_le(s, 0x30000);
out_uint32_le(s, 0x050000);
out_uint32_le(s, 2);
out_uint32_le(s, 0);
```

```
    out_uint32_le(s, 0xffffffc4);
    out_uint32_le(s, 0xfffffffe);
    out_uint32_le(s, self->mod->client_info->rdp5_performanceflags);
    out_uint16_le(s, 0);
  }
  s_mark_end(s);
  if (rdp_sec_send(self->sec_layer, s, sec_flags, MCS_GLOBAL_CHANNEL) != 0)
  {
    free_stream(s);
    DEBUG(("out rdp_rdp_send_login_info error 2"));
    return 1;
  }
  free_stream(s);
  DEBUG(("out rdp_rdp_send_login_info"));
  return 0;
}


/*****************************************************************************/
int APP_CC
rdp_rdp_connect(struct rdp_rdp* self, char* ip, char* port, struct list* channel_list_param)
{
  int flags;
  char* name;
  int num_channels;
  int i;
  struct mcs_channel_item* channel_item;

  DEBUG(("in rdp_rdp_connect"));

  flags = RDP_LOGON_NORMAL;
  if (g_strlen(self->mod->password) > 0)
  {
    flags |= RDP_LOGON_AUTO;
  }
  if (rdp_sec_connect(self->sec_layer, ip, port, channel_list_param) != 0)
  {
    DEBUG(("out rdp_rdp_connect error rdp_sec_connect failed"));
    return 1;
  }
  if (rdp_rdp_send_login_info(self, flags) != 0)
  {
    fonc_log("out rdp_rdp_connect error rdp_rdp_send_login_info failed");
    DEBUG(("out rdp_rdp_connect error rdp_rdp_send_login_info failed"));
    return 1;
  }
  DEBUG(("out rdp_rdp_connect"));
  return 0;
}


/*****************************************************************************/
int APP_CC
rdp_rdp_send_input(struct rdp_rdp* self, struct stream* s,
                   int time, int message_type,
                   int device_flags, int param1, int param2)
{
  if (rdp_rdp_init_data(self, s) != 0)
  {
    return 1;
```

```
  }
  out_uint16_le(s, 1); /* number of events */
  out_uint16_le(s, 0);
  out_uint32_le(s, time);
  out_uint16_le(s, message_type);
  out_uint16_le(s, device_flags);
  out_uint16_le(s, param1);
  out_uint16_le(s, param2);
  s_mark_end(s);
  if (rdp_rdp_send_data(self, s, RDP_DATA_PDU_INPUT, MCS_GLOBAL_CHANNEL) != 0)
  {
    return 1;
  }
  return 0;
}



/*****************************************************************************/
int APP_CC
rdp_rdp_send_invalidate(struct rdp_rdp* self, struct stream* s,
                        int left, int top, int width, int height)
{
  if (rdp_rdp_init_data(self, s) != 0)
  {
    return 1;
  }
  out_uint32_le(s, 1);
  out_uint16_le(s, left);
  out_uint16_le(s, top);
  out_uint16_le(s, (left + width) - 1);
  out_uint16_le(s, (top + height) - 1);
  s_mark_end(s);
  if (rdp_rdp_send_data(self, s, 33, MCS_GLOBAL_CHANNEL) != 0)
  {
    return 1;
  }
  return 0;
}

/*****************************************************************************/
/* return error*/
int APP_CC
rdp_rdp_recv(struct rdp_rdp* self, struct stream* s, int* type)
{
  int len;
  int chan;
  int pdu_type;
  int pdu_data_type;

  chan = 0;
  DEBUG(("in rdp_rdp_recv"));

  if (s->next_packet >= s->end || s->next_packet == 0)
  {
    if (rdp_sec_recv(self->sec_layer, s, &chan) != 0)
    {
      fonc_log("Error in rdp_rdp_recv, rdp_sec_recv failed");
      DEBUG(("error in rdp_rdp_recv, rdp_sec_recv failed"));
      return 1;
```

```
    }
    if (self->version == 0xff)
    {
      s->next_packet = s->end;
      *type = 0;
      return 0;
    }
    else if (self->version != 3)
    {
      /* We must verify this condition because I'm not pretty sure that
         it's good. I think we need to recover protocol version and not
         this one. By the moment, I put the same condition that appears
         in rdesktop */
      rdp5_process_data(self, s, pdu_data_type);
      *type = 0;
      return 0;
    }
    s->next_packet = s->p;
  }
  else
  {
    s->p = s->next_packet;
  }

  in_uint16_le(s, len);
  DEBUG(("rdp_rdp_recv got %d len", len));
  if (len == 0x8000)
  {
    s->next_packet += 8;
    *type = 0;
    DEBUG(("out rdp_rdp_recv"));
    return 0;
  }
  in_uint16_le(s, pdu_type);
  in_uint8s(s, 2);
  *type = pdu_type & 0xf;
  s->next_packet += len;
  self->chan_id = chan;
  DEBUG(("out rdp_rdp_recv"));
  return 0;
}


/*****************************************************************************/
int APP_CC
rdp_rdp_process_data_pdu(struct rdp_rdp* self, struct stream* s)
{
  int data_pdu_type;
  int ctype;
  int clen;
  int len;
  int rv;
  int roff;
  int rlen;
  char text[256];

  struct stream* ns;

  make_stream(ns);
  init_stream(ns, 8192);
```

```
  fonc_log("in rdp_rdp_process_data_pdu");
  rv = 0;
  in_uint8s(s, 6); /* shareid, pad, streamid */
  in_uint16_le(s, len);
  in_uint8(s, data_pdu_type);
  in_uint8(s, ctype);
  in_uint16_le(s, clen);
  clen -= 18;

  if (ctype & RDP_MPPC_COMPRESSED)
  {
    if (len > RDP_MPPC_DICT_SIZE)
    {
      error("error decompressed packet size exceeds max\n");
    }
    if (mppc_expand(s->p, clen, ctype, &roff, &rlen) == -1)
    {
      error("error while decompressing packet\n");
    }
    ns->data = (char *) g_malloc(sizeof(ns->data), 1);

    g_memcpy((ns->data), (unsigned char *) (g_mppc_dict.hist + roff), rlen);

    ns->size = rlen;
    ns->end = (ns->data + ns->size);
    ns->p = ns->data;
    ns->rdp_hdr = ns->p;

    s = ns;
  }
  switch (data_pdu_type)
  {
    case RDP_DATA_PDU_UPDATE:
      rv = rdp_rdp_process_update_pdu(self, s);
      break;
    case RDP_DATA_PDU_CONTROL:
      break;
    case RDP_DATA_PDU_SYNCHRONISE:
      break;
    case RDP_DATA_PDU_POINTER:
      rv = rdp_rdp_process_pointer_pdu(self, s);
      break;
    case RDP_DATA_PDU_BELL:
      break;
    case RDP_DATA_PDU_LOGON:
      break;
    case RDP_DATA_PDU_DISCONNECT:
      rv = rdp_rdp_process_disconnect_pdu(self, s);
      break;
    default:
      fonc_log("In rdp_rdp_process_data_pdu --> processing default data PDU");
      break;
  }
  return rv;
}


/*****************************************************************************/
/* Process a bitmap capability set */
```

```
static void APP_CC
rdp_rdp_process_general_caps(struct rdp_rdp* self, struct stream* s)
{
  int extraflags;

  in_uint8s(s, 10);
  /* Receiving rdp_5 extra flags supported for RDP 5.0 and later versions*/
  in_uint16_le(s, extraflags);
  if (extraflags == 0)
  {
    self->use_rdp5 = 0;
  }
}


/*****************************************************************************/
/* Send a control PDU */
/* returns error */
static int APP_CC
rdp_rdp_send_control(struct rdp_rdp* self, struct stream* s, int action)
{
  if (rdp_rdp_init_data(self, s) != 0)
  {
    return 1;
  }
  out_uint16_le(s, action);
  out_uint16_le(s, 0); /* userid */
  out_uint32_le(s, 0); /* control id */
  s_mark_end(s);
  if (rdp_rdp_send_data(self, s, RDP_DATA_PDU_CONTROL, MCS_GLOBAL_CHANNEL) !=
0)
  {
    return 1;
  }
  return 0;
}

/*****************************************************************************/
/* Send a synchronisation PDU */
/* returns error */
static int APP_CC
rdp_rdp_send_synchronise(struct rdp_rdp* self, struct stream* s)
{
  if (rdp_rdp_init_data(self, s) != 0)
  {
    return 1;
  }
  out_uint16_le(s, 1); /* type */
  out_uint16_le(s, 1002);
  s_mark_end(s);
  if (rdp_rdp_send_data(self, s, RDP_DATA_PDU_SYNCHRONISE,
MCS_GLOBAL_CHANNEL) != 0)
  {
    return 1;
  }
  return 0;
}

/*****************************************************************************/
/* Send an (empty) font information PDU */
```

```
static int APP_CC
rdp_rdp_send_fonts(struct rdp_rdp* self, struct stream* s, int seq)
{
  if (rdp_rdp_init_data(self, s) != 0)
  {
    return 1;
  }
  out_uint16_le(s, 0); /* number of fonts */
  out_uint16_le(s, 0); /* pad? */
  out_uint16_le(s, seq); /* unknown */
  out_uint16_le(s, 0x32); /* entry size */
  s_mark_end(s);
  if (rdp_rdp_send_data(self, s, RDP_DATA_PDU_FONT2, MCS_GLOBAL_CHANNEL) != 0)
  {
    return 1;
  }
  return 0;
}

/*****************************************************************************/
/* Respond to a demand active PDU */
int APP_CC
rdp_rdp_process_demand_active(struct rdp_rdp* self, struct stream* s)
{
  int type;
  int len_src_descriptor;
  int len_combined_caps;
  int channel_id;
  struct list* channel_list_param;

  in_uint32_le(s, self->share_id);
  in_uint16_le(s, len_src_descriptor);
  in_uint16_le(s, len_combined_caps);
  in_uint8s(s, len_src_descriptor);
  rdp_rdp_process_server_caps(self, s, len_combined_caps);
  rdp_rdp_send_confirm_active(self, s);
  rdp_rdp_send_synchronise(self, s);
  rdp_rdp_send_control(self, s, RDP_CTL_COOPERATE);
  rdp_rdp_send_control(self, s, RDP_CTL_REQUEST_CONTROL);
  rdp_rdp_recv(self, s, &type); /* RDP_PDU_SYNCHRONIZE */
  rdp_rdp_recv(self, s, &type); /* RDP_CTL_COOPERATE */
  rdp_rdp_recv(self, s, &type); /* RDP_CTL_GRANT_CONTROL */
  rdp_rdp_send_input(self, s, 0, RDP_INPUT_SYNCHRONIZE, 0, 0, 0);

  /* Including RDP 5.0 capabilities */
  if (self->use_rdp5 != 0)
  {
    rdp_rdp_enum_bmpcache2(self);
    rdp_rdp_send_fonts(self, s, 3);
  }
  else
  {
    rdp_rdp_send_fonts(self, s, 1);
    rdp_rdp_send_fonts(self, s, 2);
  }
  rdp_rdp_recv(self, s, &type); /* RDP_PDU_UNKNOWN 0x28 (Fonts?) */
  rdp_orders_reset_state(self->orders);
  return 0;
}
```

```
/*****************************************************************************/

int APP_CC
rdp_rdp_send_redirect_pdu(struct rdp_rdp* self, long param1, long param2,
                          long param3, int param4)
{
  char* data;
  char* name;
  int chan_id;
  int total_data_length;
  int size;
  int flags;
  int index;
  int channel_id;
  int num_channels_src;
  int num_channels_dst;
  int sec_flags;
  struct stream* s_out;
  struct mcs_channel_item* channel_item;

  DEBUG(("in rdp_rdp_send_redirect_pdu"));
  fonc_log("In rdp_rdp_send_redirect ---> processing channel redirection info");

  /* We need to verify this in order to right process the stream passed */
  chan_id = (int)(param1 & 0xffff);
  chan_id = chan_id + MCS_GLOBAL_CHANNEL + 1;
  flags = (int)((param1 >> 16) & 0xffff);
  size = param2;
  data = (char*)param3;
  total_data_length = param4;

  /* We need to recover the name of the channel linked with this
     channel_id in order to match it with the same channel on the
     first channel_list created by the RDP client at initialization
     process*/

  num_channels_src = self->mod->channel_list->count;
  for (index = 0; index < num_channels_src; index++)
  {
    channel_item = (struct mcs_channel_item*)
                   list_get_item(self->mod->channel_list, index);
    if (chan_id == channel_item->chanid)
    {
      name = channel_item->name;
    }
  }

  /* Here, we're going to search the correct channel in order to send
     information throughout this channel to RDP server */

  num_channels_dst = self->sec_layer->mcs_layer->channel_list->count;
  for (index = 0; index < num_channels_dst; index++)
  {
    channel_item = (struct mcs_channel_item*)
                   list_get_item(self->sec_layer->mcs_layer->channel_list, index);
    if (strcmp(name, channel_item->name) == 0)
    {
      channel_id = channel_item->chanid;
    }
  }
```

```
  printf("in rdp_rdp_send_redirect_pdu \n");
  g_hexdump(data, total_data_length);

    /*Copy data from s to data and after that close stream and send
    it to rdp_rdp_send_data with channel_id so we need to pass chan_id to
    rdp_rdp_send_data also in order to be able to redirect data in the correct
    way*/

  make_stream(s_out);
  init_stream(s_out, 8192);

  if (rdp_channel_init(self, s_out) != 0)
  {
    fonc_log("in rdp_rdp_send_redirect_pdu, rdp_rdp_init_data failed!!");
    return 1;
  }
  s_pop_layer(s_out, channel_hdr);

  out_uint32_le(s_out, total_data_length);
  out_uint32_le(s_out, flags);
  out_uint8a(s_out, data, size);
  s_mark_end(s_out);
  printf("in rdp_rdp_send_redirect_pdu, sending data from s->p \n");
  g_hexdump(s_out->p, size + 8);
  sec_flags = SEC_ENCRYPT;

  if(total_data_length > CHANNEL_CHUNK_LENGTH)
  {
    fonc_log(" Length exceeded in rdp_rdp_send_redirect_pdu");
  }

  /* We need to call rdp_rdp_send_data but with another code because we need to build an
     virtual_channel packet and not an MCS_GLOBAL_CHANNEL packet */

  if (rdp_sec_send(self->sec_layer, s_out, sec_flags, channel_id) != 0)
  {
    fonc_log("Error sending information to RDP server");
    return 1;
  }
  free_stream(s_out);
  fonc_log("Out of rdp_rdp_send_redirect info --> redirection info processed succesfully");
  return 0;
}

/*****************************************************************************/
int APP_CC
rdp5_process_data(struct rdp_rdp* self, struct stream* s, int type)
{

  int length;
  int roff;
  int rlen;
  int count;
  int rv;
  int x;
  int y;
  char ctype;
  char* next;
```

```
struct stream* ts;
struct stream* ns;

make_stream(ts);
init_stream(ts, 8192);
make_stream(ns);
init_stream(ns, 8192);
next = NULL;

while (s->p < s->end)
{
  in_uint8(s, type);
  if (type & RDP5_COMPRESSED)
  {
     in_uint8(s, ctype);
     in_uint16_le(s, length);
     type ^= RDP5_COMPRESSED;
  }
  else
  {
    ctype = 0;
    in_uint16_le(s, length);
  }
  next = s->p + length;

  if (ctype & RDP_MPPC_COMPRESSED)
  {
    /* TODO: it's required to create a function in the XRDP module supporting
       MPPC compression */

    if(mppc_expand(s->p, length, ctype, &roff, &rlen) == -1)
    {
      error("error while decompressing packet\n");
    }
    ns->data = (char*)g_malloc(sizeof(ns->data),1);
    g_memcpy((ns->data), (unsigned char*)(g_mppc_dict.hist +roff), rlen);
    ns->size = rlen;
    ns->end = (ns->data + ns->size);
    ns->p = ns->data;
    ns->rdp_hdr = ns->p;

    ts = ns;
  }
  else
  {
    ts = s;
  }
  switch (type)
  {
     case 0:      /* update orders */
        in_uint16_le(ts, count);
        rdp_orders_process_orders(self->orders, ts, count);
        break;
     case 1:      /* update bitmap */
        in_uint8s(ts, 2);  /* part length */
        rdp_rdp_process_bitmap_updates(self, ts);
        break;
     case 2:      /* update palette */
        in_uint8s(ts, 2);  /* uint16 = 2 */
        rdp_rdp_process_palette(self, ts);
```

```
            break;
        case 3:     /* update synchronize */
            break;
        case 5:     /* null pointer */
            break;
        case 6:     /* default pointer */
            break;
        case 8:     /* pointer position */
            in_uint16_le(ts, x);
            in_uint16_le(ts, y);
            break;
        case 9:     /* color pointer */
            rv = rdp_rdp_process_color_pointer_pdu(self, ts);
            break;
        case 10:    /* cached pointer */
            rv = rdp_rdp_process_cached_pointer_pdu(self, ts);
            break;
        default:
            unimpl("RDP5 opcode %d\n", type);
    }
    s->p = next;
  }
  return 0;
}

/*****************************************************************************/
struct list* APP_CC
list_create(void)
{
  struct list* self;

  self = (struct list*)g_malloc(sizeof(struct list), 1);
  self->grow_by = 10;
  self->alloc_size = 10;
  self->items = (tbus*)g_malloc(sizeof(tbus) * 10, 1);
  return self;
}

/*****************************************************************************/
void APP_CC
list_delete(struct list* self)
{
  int i;

  if (self == 0)
  {
    return;
  }
  if (self->auto_free)
  {
    for (i = 0; i < self->count; i++)
    {
      g_free((void*)self->items[i]);
      self->items[i] = 0;
    }
  }
  g_free(self->items);
  g_free(self);
}
```

```
/*****************************************************************************/
int APP_CC
rdp_rdp_out_bmpcache2_caps(struct rdp_rdp* self, struct stream* s)
{
  int i;

  out_uint16_le(s, RDP_CAPSET_BMPCACHE2);
  out_uint16_le(s, RDP_CAPLEN_BMPCACHE2);

  out_uint16_le(s, self->mod->client_info->bitmap_cache_persist_enable ? 2 : 0);           /*
version */

  out_uint16_be(s, 3);    /* number of caches in this set */

  /* Sending bitmap capabilities version 2 */
  i = self->mod->client_info->cache1_entries;
  i = MIN(i, 2000);
  out_uint32_le(s, i);
  i = self->mod->client_info->cache2_entries;
  i = MIN(i, 2000);
  out_uint32_le(s, i);
  i = self->mod->client_info->cache3_entries;
  i = MIN(i, 2000);
  out_uint32_le(s, i);

  out_uint8s(s, 20);        /* other bitmap caches not used */
}

/*****************************************************************************/

/* Send persistent bitmap cache enumeration PDU's
   Not implemented yet because it should be implemented
   before in xrdp_rdp_process_data case. The problem is that
   we don't save the bitmap key list attached with rdp_bmpcache2 capability
   message so we can't develop this function yet */

static void
rdp_rdp_enum_bmpcache2(struct rdp_rdp* self)
{

}

/*****************************************************************************/
int APP_CC
mppc_expand(char* data, int clen, int ctype, int* roff, int* rlen)
{
  int i;
  int k;
  int next_offset;
  int old_offset;
  int match_off;
  int walker;
  int walker_len;
  int match_len;
  int match_bits;
  int big;
  char* dict;

  dict = g_mppc_dict.hist;
  walker_len = 0;
```

```
match_len = 0;
big = ctype & RDP_MPPC_BIG ? 1 : 0;

if ((ctype & RDP_MPPC_COMPRESSED) == 0)
{
   *roff = 0;
   *rlen = clen;
   return 0;
}
if ((ctype & RDP_MPPC_FLUSH) == 0)
{
  g_memset(dict, 0, RDP_MPPC_DICT_SIZE);
  g_mppc_dict.roff = 0;
}


if ((ctype & RDP_MPPC_RESET) == 0)
{
  g_mppc_dict.roff = 0;
}

*roff = 0;
*rlen = 0;

walker = g_mppc_dict.roff;
old_offset = next_offset;
*roff = old_offset;

if (clen == 0)
{
  return 0;
}

clen += i;

do
{
  if (walker_len == 0)
  {
    if (i >= clen)
    {
      break;
    }
  walker = data[i++] << 24;
  walker_len = 8;
  }
  if (walker >= 0)
  {
    if (walker_len < 8)
    {
      if (i >=clen)
      {
        if (walker != 0)
        {
          return -1;
        }
        break;
      }
    walker |= (data[i++] & 0xff) << (24 - walker_len);
    walker_len += 8;
```

```
  }
  if (next_offset >= RDP_MPPC_DICT_SIZE)
  {
    return -1;
  }
  dict[next_offset++] = (walker >> 24);
  walker <<= 8;
  walker_len -=8;
  continue;
}
walker <<= 1;
if (--walker_len < (big ? 3 : 2))
{
  if (i > clen)
  {
    return -1;
  }
  walker |= (data[i++] & 0xff) << (24 - walker_len);
  walker_len += 8;
}
if (big)
{
  /* Here we're going to process Copy offset in order to decode it,
     depending of MPPC algorithme (8k or 64k). In this point we're
     processing MPPC-64k compressed packets */

  switch (walker >> 29)
  {
    case 7: /* Copy offset Range < 63 */
            for (; walker_len < 9; walker_len += 8)
            {
              if (i >= clen)
              {
                return -1;
              }
            walker |= (data[i++] & 0xff) << (24 - walker_len);
            }
            walker <<= 3;
            match_off = walker >> 26;
            walker <<= 6;
            walker_len -= 9;
            break;
    case 6: /* Copy offset range between 64 and 319 */
            for (;walker_len < 11; walker_len += 8)
            {
              if (i >= clen)
              {
                return -1;
              }
            walker |= (data[i++] & 0xff) << (24 - walker_len);
            }
            walker <<= 3;
            match_off = (walker >> 24) + 64;
            walker <<= 8;
            walker_len -= 11;
            break;
    case 5:
    case 4: /* Copy offset range between 320 and 2367 */
           for (; walker_len < 13; walker_len += 8)
           {
```

```
            if (i >= clen)
            {
              return -1;
            }
           walker |= (data[i++] & 0xff) << (24 - walker_len);
          }
          walker <<= 2;
          match_off = (walker >> 21) + 320;
          walker <<= 11;
          walker_len -= 13;
          break;
    default: /* Copy offset range between 2368 and 65535 */
           for (; walker_len < 17; walker_len += 8)
           {
            if (i >= clen)
            {
              return -1;
            }
           walker |= (data[i++] & 0xff) << (24 - walker_len);
           }
           walker <<= 1;
           match_off = (walker >> 16) + 2368;
           walker_len <<= 16;
           walker_len -= 17;
           break;
  }
}
else
{
    /* MPPC-8K Compression used */
  switch (walker >> 30)
  {
    case 3: /* Copy offset < 63 */
          if (walker_len < 8)
          {
            if (i>=clen)
            {
              return -1;
            }
            walker |= (data[i++] & 0xff) << (24 - walker_len);
            walker_len += 8;
          }
          walker <<= 2;
          match_off = walker >> 26;
          walker <<= 6;
          walker_len -=8;
          break;
    case 2: /* Copy offset range between 64 and 319 */
          for (; walker_len < 10; walker_len += 8)
          {
            if (i >= clen)
            {
              return -1;
            }
            walker |= (data[i++] & 0xff) << (24 - walker_len);
          }
          walker <<= 2;
          match_off = (walker >> 24) + 64;
          walker <<= 8;
          walker_len -= 10;
```

```
            break;
    default: /*Copy offset range between 320 and 8191 */
             for(; walker_len < 14; walker_len += 8)
             {
               if (i>= clen)
               {
                 return -1;
               }
               walker |= (data[i++] & 0xff) << (24 - walker_len);
             }
             match_off = (walker >> 18) + 320;
             walker <<= 14;
             walker_len -= 14;
             break;
  }
}
if (walker_len == 0)
{
  if (i >= clen)
  {
    return 1;
  }
  walker = data[i++] << 24;
  walker_len = 8;
}

/* Decode Length of Match */
if (walker >= 0)
{
  match_len = 3;
  walker <<= 1;
 walker_len--;
}
else
{
  /* Decoding L o M values according with the table in RDPBCGR at page 215 */
  match_bits = big ? 14 : 11;
  do
  {
    walker <<= 1;
    if (--walker_len == 0)
    {
      if (i >=clen)
      {
        return -1;
      }
    walker = data[i++] << 24;
    walker_len = 8;
    }
    if (walker >= 0)
    {
      break;
    }
    if (--match_bits == 0)
    {
      return -1;
    }
  }
  while(1);
  match_len = (big ? 16 : 13) - match_bits;
```

```
      walker <<= 1;
      if (--walker_len < match_len)
      {
        for (; walker_len < match_len; walker_len += 8)
        {
          if (i >= clen)
          {
            return -1;
          }
          walker |= (data[i++] & 0xff) << (24 - walker_len);
        }
      }

      match_bits = match_len;
      match_len = ((walker >> (32 - match_bits)) &
                   (~(-1 << match_bits))) | (1 << match_bits);
      walker <<= match_bits;
      walker_len -= match_bits;
    }
    if (next_offset + match_len >= RDP_MPPC_DICT_SIZE)
    {
      return -1;
    }
    k = (next_offset - match_off) & (big ? 65535 : 8191);
    do
    {
      dict[next_offset++] = dict[k++];
    }
    while (--match_len != 0);
  }
  while(1);

  g_mppc_dict.roff =  next_offset;

  *roff = old_offset;
  *rlen = next_offset - old_offset;
  return 0;
}

/*****************************************************************************/
 /* Returns error */
int APP_CC
rdp_channel_init(struct rdp_rdp* self, struct stream* s)
{
  if (rdp_sec_init(self->sec_layer, s, SEC_ENCRYPT) != 0)
  {
    return 1;
  }
  s_push_layer(s, channel_hdr, 8);
  return 0;
}
```

Xrdp/rdp/rdp_sec.c

```
/*****************************************************************************/
/* returns error */
int APP_CC
rdp_sec_recv(struct rdp_sec* self, struct stream* s, int* chan)
{
  uint32_t flags;
  uint32_t length;
  int channel_flags;
  char swapbyte;
  char text[256];

  DEBUG((" in rdp_sec_recv"));
  if (rdp_mcs_recv(self->mcs_layer, s, chan) != 0)
  {
    DEBUG((" error in rdp_sec_recv, rdp_mcs_recv failed"));
    fonc_log("error in rdp_sec_recv, rdp_mcs_recv failed");
    return 1;
  }
  if (self->rdp_layer->version != 3)
  {
    if (self->rdp_layer->version & 0x80)
    {
      in_uint8s(s, 8);
      rdp_sec_decrypt(self, s->p, s->end - s->p);
    }
    return 0;
  }
  in_uint32_le(s, flags);
  DEBUG((" rdp_sec_recv flags %8.8x", flags));
  if (flags & SEC_ENCRYPT)
  {
    in_uint8s(s, 8);
    rdp_sec_decrypt(self, s->p, s->end - s->p);
  }
  if (flags & SEC_LICENCE_NEG)
  {
    DEBUG((" in rdp_sec_recv, got SEC_LICENCE_NEG"));
    rdp_lic_process(self->lic_layer, s);
    *chan = 0;
  }

  if (flags & 0x0400)
  {
    in_uint8s(s, 8);
    rdp_sec_decrypt(self, s->p, s->end - s->p);


    if (s->p[0] == 0 && s->p[1] == 4)
    {

        swapbyte = s->p[0];
        s->p[0] = s->p[2];
        s->p[2] = swapbyte;

        swapbyte = s->p[1];
        s->p[1] = s->p[3];
        s->p[3] = swapbyte;
```

```
        swapbyte = s->p[2];
        s->p[2] = s->p[3];
        s->p[3] = swapbyte;
      }
    }
    if (*chan != MCS_GLOBAL_CHANNEL)
    {
      if (*chan > 0)
      {
        in_uint32_le(s, length);
        in_uint32_le(s, channel_flags);
        rdp_process_redirect_pdu(self, s, channel_flags, length, *chan);
      }
      self->rdp_layer->version = 0xff;
      return 0;
    }
    DEBUG((" out rdp_sec_recv"));
    return 0;
}

/*****************************************************************************/
/* prepare client mcs data to send in mcs layer */
static void APP_CC
rdp_sec_out_mcs_data(struct rdp_sec* self, struct list* channel_list_param)
{
    struct stream* s;
    struct mcs_channel_item* channel_item;
    int hostlen;
    int length;
    int num_channels;
    int flags;
    int i;
    char* name;

    num_channels = channel_list_param->count;

    s = self->client_mcs_data;
    init_stream(s, 512);
    self->rdp_layer->mod->hostname[15] = 0; /* limit length to 15 */
    hostlen = 2 * g_strlen(self->rdp_layer->mod->hostname);
    length = 158 + 76 + 12 + 4;
    if (num_channels > 0)
    {
       length += (num_channels * 12) + 8;
    }
    /* Added in order to limit hostlen and hostname size */
    if (hostlen > 30)
    {
      hostlen = 30;
    }

    /* Generic Conference Control (T.124) ConferenceCreateRequest */
    out_uint16_be(s, 5);
    out_uint16_be(s, 0x14);
    out_uint8(s, 0x7c);
    out_uint16_be(s, 1);
    out_uint16_be(s, (length | 0x8000)); /* remaining length */
    out_uint16_be(s, 8); /* length? */
    out_uint16_be(s, 16);
    out_uint8(s, 0);
```

```
  out_uint16_le(s, 0xc001);
  out_uint8(s, 0);
  out_uint32_le(s, 0x61637544); /* OEM ID: "Duca", as in Ducati. */
  out_uint16_be(s, ((length - 14) | 0x8000)); /* remaining length */

  /* Client information */
  out_uint16_le(s, SEC_TAG_CLI_INFO);
  out_uint16_le(s, 212); /* length */
  //out_uint16_le(s, self->rdp_layer->use_rdp5 ? 4 : 1); /* RDP version. 1 == RDP4, 4 == RDP5.
*/
  out_uint16_le(s, 4); /* We've forcing to run in RDP5 */
  out_uint16_le(s, 8);
  out_uint16_le(s, self->rdp_layer->mod->width);
  out_uint16_le(s, self->rdp_layer->mod->height);
  out_uint16_le(s, 0xca01);
  out_uint16_le(s, 0xaa03);
  out_uint32_le(s, self->rdp_layer->mod->keylayout);
  out_uint32_le(s, 2600); /* Client build */

  /* Unicode name of client, padded to 32 bytes */
  rdp_rdp_out_unistr(s, self->rdp_layer->mod->hostname);
  out_uint8s(s, 30 - hostlen);
  out_uint32_le(s, 4); /* IBM enhanced 101-102 keys keyboard */
  out_uint32_le(s, 0);/* keyboard subtype */
  out_uint32_le(s, 12);/* Number of function keys */
  out_uint8s(s, 64); /* reserved? 4 + 12 doublewords */
  out_uint16_le(s, 0xca01); /* color depth? */
  out_uint16_le(s, 1);
  out_uint32_le(s, 0);
  out_uint8(s, self->rdp_layer->mod->rdp_bpp);
  out_uint16_le(s, 0x0700);
  out_uint8(s, 0);
  out_uint32_le(s, 1);
  out_uint8s(s, 64); /* End of client info */

  out_uint16_le(s, SEC_TAG_CLI_4);
  out_uint16_le(s, 12);
  out_uint32_le(s, 9);
  out_uint32_le(s, 0);

  /* Client encryption settings */
  out_uint16_le(s, SEC_TAG_CLI_CRYPT);
  out_uint16_le(s, 12); /* length */
  out_uint32_le(s, 0x3); /* encryption supported, 128-bit supported */
  out_uint32_le(s, 0); /* Unknown */

  /*Here we need to put channel information in order to redirect channel data
  from client to server passing through the "proxy" */

  if (channel_list_param->count > 0)
  {
    out_uint16_le(s, SEC_TAG_CLI_CHANNELS);
    out_uint16_le(s, num_channels * 12 + 8);
    out_uint32_le(s, num_channels);
    for (i = 0; i < num_channels; i++)
    {
       channel_item = (struct mcs_channel_item*)
                  list_get_item(channel_list_param, i);
       name = channel_item->name;
       flags = channel_item->flags;
```

```
      out_uint8a(s, name, 8);
      out_uint32_be(s, flags);
    }
  }
  s_mark_end(s);
}


/*****************************************************************************/
/* Process connect response data blob */
int APP_CC
rdp_sec_process_mcs_data(struct rdp_sec* self, struct stream* s)
{
  int tag;
  int length;
  int len;
  char* next_tag;
  //struct stream* s;

  /*s = self->server_mcs_data;
  s->p = s->data;*/

  in_uint8s(s, 21); /* header (T.124 ConferenceCreateResponse) */
  in_uint8(s, len);
  if (len & 0x80)
  {
    in_uint8(s, len);
  }
  while (s->p < s->end)
  {
    in_uint16_le(s, tag);
    in_uint16_le(s, length);
    DEBUG((" rdp_sec_process_mcs_data tag %d length %d", tag, length));
    if (length <= 4)
    {
      return;
    }
    next_tag = (s->p + length) - 4;
    switch (tag)
    {
      case SEC_TAG_SRV_INFO:
        //rdp_sec_process_srv_info(self, s);
        break;
      case SEC_TAG_SRV_CRYPT:
        rdp_sec_process_crypt_info(self, s);
        break;
      case SEC_TAG_SRV_CHANNELS:
        rdp_sec_process_srv_channels(self, s);
        break;
      default:
        DEBUG((" In rdp_sec_process_mcs_data, unimplemented tag"));
        break;
    }
    s->p = next_tag;
  }
  return 0;
}

/*****************************************************************************/
```

```
/* Establish a secure connection */
int APP_CC
rdp_sec_connect(struct rdp_sec* self, char* ip, char* port, struct list* channel_list_param)
{

  DEBUG((" in rdp_sec_connect"));
  rdp_sec_out_mcs_data(self, channel_list_param);
  if (rdp_mcs_connect(self->mcs_layer, ip, port, channel_list_param) != 0)
  {
    DEBUG((" out rdp_sec_connect error rdp_mcs_connect failed"));
    return 1;
  }
  if (rdp_sec_establish_key(self) != 0)
  {
    fonc_log(" out rdp_sec_connect error rdp_sec_establish_key failed");
    DEBUG((" out rdp_sec_connect error rdp_sec_establish_key failed"));
    return 1;
  }
  DEBUG((" out rdp_sec_connect"));
  return 0;
}

/*****************************************************************************/
/* returns error */
int APP_CC
rdp_sec_send(struct rdp_sec* self, struct stream* s, int flags, int chan_id)
{
  int datalen;
  char text[256];

  DEBUG((" in rdp_sec_send flags %8.8x", flags));
  s_pop_layer(s, sec_hdr);
  out_uint32_le(s, flags);
  if (chan_id != MCS_GLOBAL_CHANNEL)
  {
    printf("in rdp_sec_send, sending info throughout channel %d \n", chan_id);
    printf("in rdp_sec_send, sending data from s->p \n");
    g_hexdump(s->p,(s->end - s->p));
  }
  if (flags & SEC_ENCRYPT)
  {
    datalen = (s->end - s->p) - 8;
    rdp_sec_sign(s->p, 8, self->sign_key, self->rc4_key_len, s->p + 8,
                 datalen);
    rdp_sec_encrypt(self, s->p + 8, datalen);
    if (chan_id != MCS_GLOBAL_CHANNEL)
    {
      if (datalen > CHANNEL_CHUNK_LENGTH)
      {
        fonc_log(" Out of rdp_mcs_send, length exceeded: ");
        g_snprintf(text, 255, "%d", datalen);
        fonc_log(text);
      }
    }
  }
  if (rdp_mcs_send(self->mcs_layer, s, chan_id) != 0)
  {
    DEBUG((" out rdp_sec_send, rdp_mcs_send failed"));
    return 1;
  }
```

```
  DEBUG((" out rdp_sec_send"));
  return 0;
}

/*****************************************************************************/
/* this adds the mcs channels in the list of channels to be used when
   creating the server mcs data */

static void APP_CC
rdp_sec_process_srv_channels(struct rdp_sec* self, struct stream* s)
{
  int num_channels;
  int base_channel;
  int index;
  int chanid;
  struct mcs_channel_item* channel_item_cli;
  struct mcs_channel_item* channel_item_srv;

  DEBUG(("processing channels, channel_code is %d", self->channel_code));

  /* this is an option set in xrdp.ini, use 1 by default, static virtual
  channels accepted */
  if (self->channel_code != 1) /* are channels on? */
  {
    return;
  }

  in_uint16_le(s, base_channel);
  in_uint16_le(s, num_channels);

  /* We assume that the channel_id array is confirmed in the same order
  that it has been sent. If there are any channels not confirmed, they're
  going to be the last channels on the array sent in MCS Connect Initial */

  for (index = 0; index < num_channels; index++)
  {
    channel_item_cli = (struct mcs_channel_item*)
           list_get_item(self->rdp_layer->mod->channel_list, index);

    channel_item_srv = (struct mcs_channel_item*)
           g_malloc(sizeof(struct mcs_channel_item), 1);
    in_uint16_le(s, chanid);
    channel_item_srv->chanid = chanid;
    g_strcpy(channel_item_srv->name, channel_item_cli->name);
    channel_item_srv->flags = channel_item_cli->flags;
    list_add_item(self->mcs_layer->channel_list, (long)channel_item_srv);
  }
}

/*****************************************************************************/

static void APP_CC
rdp_sec_process_srv_info(struct rdp_sec* self, struct stream* s)
{
   int client_requested_protocols;
   int rdp_version;

   in_uint16_le(s, rdp_version);

   if (rdp_version == 1)
```

```
  {
    self->rdp_layer->use_rdp5 = 0;
    self->rdp_layer->mod->rdp_bpp = 8;
  }
  else
  {
   self->rdp_layer->use_rdp5 = 1;
  }
}


/*****************************************************************************/
/*Added for Javier Caverni in order to deploy channel redirection in the rdp
module of xrdp*/

int APP_CC
rdp_process_redirect_pdu(struct rdp_sec* self, struct stream* s, int flags, int length, int chanid)
{

   int data_length;
   int total_length;
   int num_channels_src;
   int num_channels_dst;
   int index;
   int channel_id;
   int size;
   char* name;
   struct mcs_channel_item* channel_item;

   /* We need to recover the name of the channel linked with this
    channel_id in order to match it with the same channel on the
    first channel_list created by the RDP client at initialization
    process*/


   num_channels_src = self->mcs_layer->channel_list->count;
   for (index = 0; index < num_channels_src; index++)
   {
     channel_item = (struct mcs_channel_item*)
                 list_get_item(self->mcs_layer->channel_list, index);
     if (chanid == channel_item->chanid)
     {
       name = channel_item->name;
     }
   }

   /* Here, we're going to search the correct channel in order to send
   information throughout this channel to RDP client*/

   num_channels_dst = self->rdp_layer->mod->channel_list->count;
   for (index = 0; index < num_channels_dst; index++)
   {
     channel_item = (struct mcs_channel_item*)
                 list_get_item(self->rdp_layer->mod->channel_list, index);
     if (strcmp(name, channel_item->name) == 0)
     {
       channel_id = channel_item->chanid;
     }
   }
   size = (int)(s->end - s->p);
```

```
   data_length = size;
   printf("Recovering size of data received: %d \n", size);
   printf("Receiving data from channel: %s with length: %d \n", name, length);
   g_hexdump(s->p, length);

   if (channel_id >= 0)
   {
     self->rdp_layer->mod->server_send_to_channel(self->rdp_layer->mod,
                              channel_id, s->p, data_length, length, flags);
   }
   else
   {
     fonc_log("Error sending information, wrong channel id");
   }
   DEBUG((" In rdp_process_redirect_pdu, sending info"));

   fonc_log("In rdp_process_redirect_pdu --> redirect info processed");
   return 0;
}

/*****************************************************************************/

/* Parse a public signature structure */
int APP_CC
rdp_sec_parse_public_sig(struct rdp_sec* self, struct stream* s, int len, char* modulus, char*
exponent)
{
        char signature[SEC_MAX_MODULUS_SIZE];
        int sig_len;

        if (len != 72)
        {
          fonc_log("in rdp_sec_parse_public_sig, len > 72");
          DEBUG(("in rdp_sec_parse_public_sig, len > 72"));
    return 0;
        }
        g_memset(signature, 0, sizeof(signature));
        sig_len = len - 8;
        in_uint8a(s, signature, sig_len);
        return ssl_sig_ok(exponent, SEC_EXPONENT_SIZE, modulus, self-
>server_public_key_len,
                                signature, sig_len);
}
```