Journal of Software Engineering and Applications, **2011, 4, 379-387**
doi:10.4236/jsea.2011.46043 Published Online June 2011 (http://www.SciRP.org/journal/jsea)

Stack E6 and Its Implementation within Linux Kernel

Dmitry Anatoly Zaitsev, Kyril Dmitry Guliaiev

Department of Computer Science, International Humanitarian University, Odessa, Ukraine.
Email: zsoftua@yahoo.com, k.guliaiev@gmail.com

Received February 5th, 2010; revised March 18th, 2011; accepted April 16th, 2011.

ABSTRACT

The first implementation of new E6 stack of networking protocols within the kernel of an operating system is presented. Stack E6 was developed to increase the efficiency of a network entirely built on the base of Ethernet technology. It uses a uniform hierarchical E6 address on all the levels and annuls TCP, UDP and IP protocols. The experimental implementation adds a new system call to the kernel of Linux and a new type of Ethernet E6 frame. All the application interface standards are saved according to RFC except of E6 address usage instead of IP address and instead of Ethernet MAC address as well.

Keywords: *Stack of Protocols, E6, Linux, Kernel, System Call, Ethernet*

1. Introduction

Stack of protocols E6 was presented in [1] then the corresponding patent of Ukraine on utility model [2] was obtained and the robustness of E6 networks was acknowledged [3] via modeling in CPN Tools. The next steps assume its implementation within real-life operating systems and the development of special hardware namely switching routers E6 (SRE6). On the periphery of E6 network usual Ethernet switches could be employed but the maximal efficiency supposes special hardware usage.

Linux kernels are open source, making implementation of the new protocol stacks possible. Different approaches could be chosen but the present implementation is developed as an independent one without the integration with Unix socket facilities. This approach is justified by the fact that Unix sockets are rather heavy and closely related with TCP/IP protocols family while the stack E6 does not use their facilities. The only requirement was to save the application interfaces providing the further adaptation of TCP/IP application software into E6 networks.

Thus the main principle of the implementation was to provide the application interface according to TCP and UDP standards [4,5] and the interface according to Ethernet standards [6] as well as independent work of stack E6 among other protocols. Ethernet frames are multiplexed/demultiplexed using the type field with new value 0xE600 which corresponds to E6 frames. The usage of TCP and UPD application interface standards [4,5] provides the possibility of the code reuse in other operating systems. So this implementation within Linux kernel could be considered rather non Unix-de pendent as far as it is not based on the Unix socket facilities.

The present experimental implementation does not employ Ethernet LLC2 facilities and provides the application interfaces of UPD protocol only. The implementation of TCP application interfaces requires Ether-net LLC2 sliding window procedures usage instead of TCP protocol procedures and is the subject for future work.

2. An Overview of E6 Technology

E6 technology [1,2] is based on two pivotal ideas: to use the same uniform hierarchical E6 address with the length of 6 bytes (**Figure 1**) on all the levels of the open systems interconnection model and to employ Ethernet LLC2 sliding window procedures instead of protocol TCP for guaranteed delivery of information.

As result only the pair of software port numbers is left to be put into the packet header (E6p2 header—**Figure 2**) and the rest of job is shifted to the Ethernet Data-Link layer by the E6 Concordance software (**Figure 3**). As for additional QoS parameters, they could be put into VLAN header of the frame. Actually E6 annuls TCP, UDP and IP protocols and the reminder is 4 bytes of the E6p2 header. It annuls also the mapping of IP addresses into

```
 0                   1                   2                   3                   4
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      E6 Nework Address              |                E6 Host Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
\                                   /
 ----------------\/-----------------

  Subnet Mask (as the number of bits of E6 network address)
```

Figure 1. The format of E6 address.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2. The E6p2 header.

```
+-------------+----------------------+----------------------+
|   OSI-ISO   |        TCP/IP        |          E6          |
+-------------+----------------------+----------------------+
| Application | HTTP, SMTP, VoIP ... | HTTP, SMTP, VoIP ... |
+-------------+-----------+----------+----------------------+
| Session     |           |          |                      |
+-------------+    TCP    +----------+                      |
| Transport   |           |    UDP   |    E6 Concordance    |
+-------------+-----------+----------+                      |
| Network     |           IP         |                      |
+-------------+----------------------+----------------------+
| Channel     |       Ethernet       |     E6 Ethernet      |
+-------------+----------------------+----------------------+
```

Figure 3. The stack of E6 protocols.

MAC addresses and corresponding ARP/RARP protocols which consumes a bulk of time on hosts and routers.

As far as E6 address is hierarchical in the same way as CIDR IP address it provides the world-wide network construction at the expense of hosts and subnets addresses aggregation under common mask that considerably reduces the address table size. Ethernet suffers from its plain MAC addresses which overflow address tables of switches because of each individual address should be listed. As far as E6 address has the same length as Ethernet MAC address (namely 6 octets) it can be put directly into the address field of the Ethernet frame instead of MAC address. Modern Ethernet adapters allow the assigning new MAC address to the interface and E6 address is assigned.

To provide an independent existence of E6 technology a new type of E6 Ethernet frames is introduced (0xE600). Hosts running E6 stack accept E6 frames while other hosts simply ignore them. Usual Ethernet switches deliver E6 frames freely because they do not analyze the frame type field usually. Traditional IP routers will drop unknown E6 packets occasionally delivered as result of broadcasting. So E6 could exist within Ethernet switched network.

To provide the scalability, special E6 switching routers (SRE6) should be developed or at least implemented as patches to the usual software of known routers. The simplest SRE6 could be created on the base of Unix computer with a few Ethernet cards.

The packet delivery scheme shown in **Figure 4** illustrates the advantages of E6 technology: the same pair of destination and source E6 addresses is constant through the entire delivery path. SRE6 only analyze the destination E6 address and switch the packet to the destination port; no additional IP-MAC addresses mapping, no packet overhead with more than 40 octets of TCP and IP headers. The computational resources of the devices are saved for higher performance and better QoS, which is especially significant for VoIP applications.

Note that, E6 technology is justified for the present headers. The computational resources of the devices are saved for higher performance and better QoS, which is especially significant for VoIP applications.

Note that, E6 technology is justified for the present

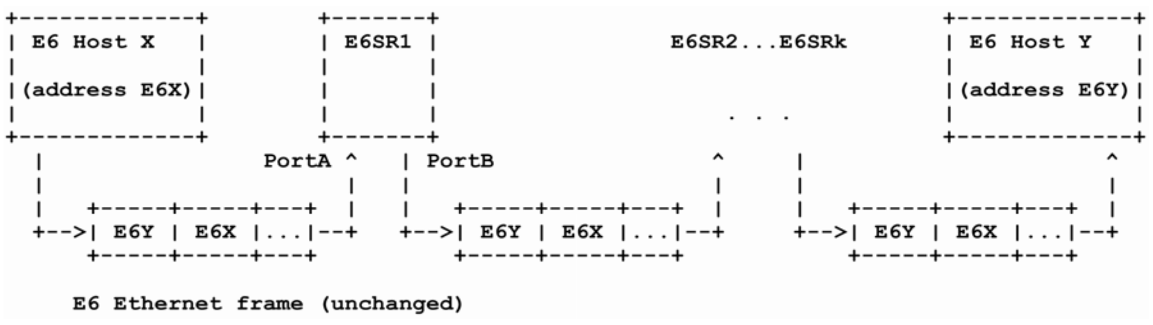

Figure 4. The E6 packet delivery scheme.

time and future tendencies — Ethernet dominates at the layer: LAN—1, 10 Gbps, campus and metropolitan networks—10 Gbps over DWDM, backbones—PBB, access networks—last-mile-Ethernet, wireless—WiFi.

3. Application Interfaces

All the code was developed as a loadable Linux kernel module E6udpmod.ko save the minor changes to the static part of the kernel with the goal to add a new system call. A conditional system call 324 with the name E6_call was added to the static part of the kernel. This conditional system call is devised to implement all the present and future E6 application interface routines. The static part of the kernel contains a pointer for the new system call 324 (which is not used in 2.6.22.9 version of the kernel) to the dummy routine sys_E6call which checks the pointer new_sys_E6call to the actual routine and calls it in the case the pointer is not equal to NULL. The pointer is initialized with the value NULL so when E6updmod.ko module is unloaded nothing occurs except sys_E6call could write messages into system log file when it is called. The pointer new_sys_E6call is exported by the kernel for the usage in modules. The details are represented in **Listing 1**.

The rest of the work is done by the module E6udpmod.ko. At initialization it sets the pointer new_ sys_E6call to the actual entry routine as well as finds Ethernet device E6_dev, copies its hardware address and registers a new E6 packet handler (**Listing 2**).

After the statement new_sys_E6call = mod_sys_E6call all the E6 system calls are handled by the routine mod_sys_E6call which is situated within E6updmod.ko and works as a dispatcher of conditional calls recognized by their numbers defined by the variable call.

The interface at the application level is provided by the library E6udplib (**Listing 3**) that contains user-end routines E6sendmsg, E6rcvmsg, E6regport, E6unregport, E6waitmsg which finally exert system call 324 using syscall routine from the standard library libc.

```
/* file syscall_table.S */
ENTRY(sys_call_table)
....
          .long sys_E6call              /* new E6 syscall 325 */

/* file socket.c   E6 new */
long (*new_sys_E6call)(int call, unsigned long __user *args) = NULL;
asmlinkage long sys_E6call(int call, unsigned long __user *args)
{
   int err;
   if( new_sys_E6call != NULL )
      err = (*new_sys_E6call)( call, args );
   else
      { printk(KERN_INFO"*** sys_E6call echo: %d\n", call); err=0; }
   return err;
}
EXPORT_SYMBOL(new_sys_E6call);
/* E6 new end */
```

Listing 1. Implementation of new system call within kernel.

```
static int E6udpmod_init_module(void)
{
          int err = 0;

          printk(KERN_INFO"*** E6udpmod: init devname=%s ***\n", E6_devname);
          /* find E6 device */
          E6_dev = dev_get_by_name(E6_devname);
          memcpy( E6_myaddr, E6_dev->dev_addr, E6_dev->addr_len );
          /* set net E6call function */
          new_sys_E6call = mod_sys_E6call;
          /* register E6 packet handler */
          dev_add_pack(&E6_packet_type);

         return 0;
}

extern long (*new_sys_E6call)(int call, unsigned long __user *args);

long mod_sys_E6call(int call, unsigned long __user *args)
{
  unsigned long a[1];
  unsigned long a0;
  unsigned char nargs = (1)*sizeof( unsigned long );
  int err;

  printk(KERN_INFO"*** E6udpmod: mod_sys_E6call %d\n", call );

  if (copy_from_user(a, args, nargs))
          return E6ERR_COPY;

  a0 = a[0];

  switch( call ){
  case E6_CALL_PUTMSG:
                   err = E6_putmsg( (struct E6msg_buf __user *)a0 );
                   break;
  case E6_CALL_GETMSG:
                   err = E6_getmsg( (struct E6msg_buf __user *)a0 );
                   break;
...
  default:
                   err = E6ERR_CALLNUM;
                   break;
  }

  return err;
}
```

Listing 2. Implementation of conditional call within loadable module.

The set of routines E6sendmsg, E6rcvmsg, E6regport, E6unregport, E6waitmsg completely satisfy the requirements of RFC 768 on application interface of UDP protocol. The interaction between the software code situated in the static part of the kernel, in the loadable module and in the user-end library is illustrated in **Figure 5**.

User interface library E6udplib was employed for the development of simple E6talk application which provides the exchange of textual messages within E6 network on the base of switched Ethernet. The application

```
#define SYS_E6call          324
#define E6_CALL_PUTMSG      1
#define E6_CALL_GETMSG      2
...
#define MAXE6BUFSIZE        1496
#define E6ADDRLEN           6

struct E6msg_buf {
        u8 E6dst_addr[E6ADDRLEN];
        u8 E6src_addr[E6ADDRLEN];
        u16 E6dst_port;
        u16 E6src_port;
        u16 E6data_len;
        u8 * E6data;
};

struct E6msg_buf buf;

int E6sendmsg( struct E6msg_buf * b, int * err )
{
 int rc;
 unsigned long a[1];

 a[0] = (unsigned long)b;

 rc = syscall( SYS_E6call, E6_CALL_PUTMSG, a );

 *err = errno;

 return rc;
}
...
```

Listing 3. E6 user-end library.

successfully provides real-life communication which acknowledges the robustness of the stack E6. To acknowledge the efficiency of E6 technology before TCP/IP protocols family a lot of work should be done yet. A traffic analyzer such as Wireshark allows the watching E6 Ethernet frames which are transmitted freely among other types of frames and do not hamper the work of other protocols. So E6 network exists in a parallel world regarding TCP/IP till the gateways between E6 and TCP/IP networks will have been done which is the direction for future work.

4. Interfaces

The interface with Ethernet hardware within Linux kernel is provided by the struct net_device which contains the device description and functions. The valid functions are given by the set of pointers which point to the actual routines of the current Ethernet driver; the basic routines are: open, stop, hard_start_xmit. The structure net_device contains the header of the output packets queue; the packet is represented by struct sk_buff * skb.

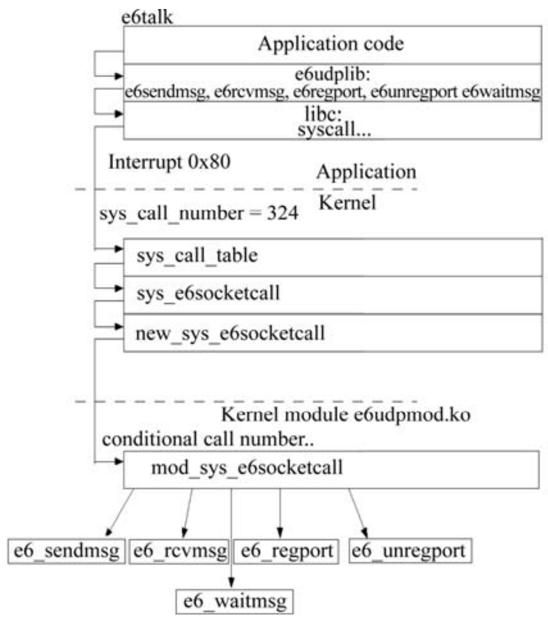

Figure 5. The interaction between the user, kernel and module software code.

```
#define ETH_P_E6                0xE600

static struct packet_type E6_packet_type = {
        .type = __constant_htons(ETH_P_E6),
        .func = E6_rcv,
        .gso_send_check = NULL, /*E6_gso_send_check,*/
        .gso_segment = NULL, /*E6_gso_segment,*/
};
dev_add_pack(&E6_packet_type);
```

Listing 4. Registering E6 packets within kernel.

The interfaces are left without changes but only used for E6 Ethernet frames transmission. The multiplexing at sending E6 packets is implemented via the new type of E6 Ethernet frames ETH_P_E6 (**Listing 4**).

The module fills in sk_buff with the user data and E6 headers using ETH_P_E6 type of frame and calls hard_start_xmit passing the buck to the Ethernet driver for the transmitting of the packet through the media.

The receiving of E6 packet is more complicated because it is done by the Ethernet driver on hardware interrupt. The driver allocates and fills in sk_buff with the arrived packet and then provides the demultiplexing procedure based on the registered types of packets. The registration of the packet types is implemented with struct packet_type and is represented in **Listing 4**.

So after the statement dev_add_pack in the module initialization routine E6udpmod_init_module (**Listing 2**), all the arrived packets of the type ETH_P_E6 are processed by E6_rcv routine situated within E6udpmod.ko module. Thus the two way interfaces with the layer are installed: for the sending E6 packets and for the receiving E6 packets. Note that, E6 packets and corresponding E6 frames are transmitted independently among other types of Ethernet frames. The scheme shown in **Figure 6** explains the interfaces of E6udpmod.ko with the Ethernet layer.

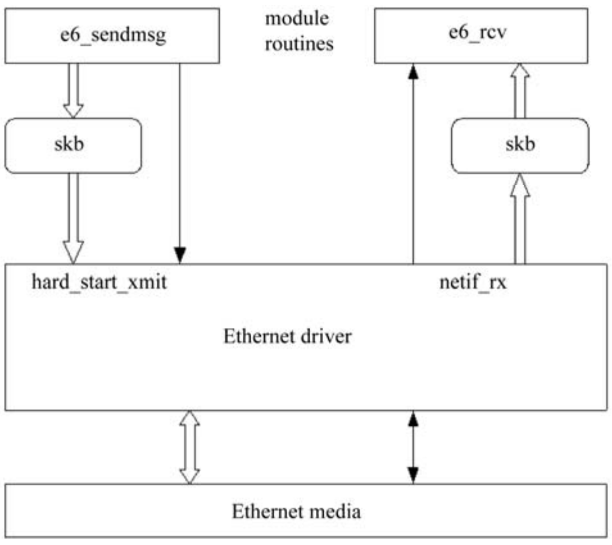

Figure 6. The interfaces with the layer.

5. Internal Data Structures and Routines

Besides the described above initialization routine E6-udpmod_init_module and the dispatcher of the conditional system call 324 with the name mod_sys_E6call the module E6udpmod.ko contains the following routines: routines for the implementation of conditional calls E6_sendmsg, E6_rcvmsg, E6_regport, E6_unregport, E6_waitmsg; the routine for the processing of arrived E6 packets E6_rcv; the module exit routine E6udpmod_cleanup_module. The interaction of the mentioned routines is shown in **Figure 5**.

Let us consider the basic data structures of the E6udpmod.ko module. Note that according to RFC 768 the source port of an application should be registered. Messages can be sent only from a registered port as well as messages can be received only to a registered port. So, the basic data structure is the list of registered ports. As the sending of a message is completely done during one system call E6sendmsg, the queues of output packets are not created within E6udpmod.ko module; sk_buff is allocated, data copied from the user address space, E6 headers created and finally sk_buff is put into the driver packet output queue via driver routine hard_start_xmit call.

At the arrival of a new E6 packet, E6_rcv routine is called by the driver and the packet is put into the appropriate queue to the registered port. Routine E6_rcv puts the packet to the tail of the queue; system call E6_rcvmsg gets the packet from the head of the queue. When the destination port is unregistered (unknown) the packet is dropped.

Thus two levels of queues are created: the queue of registered ports and the queues of received packets (to registered ports). **Figure 7** shows the interconnection of the basic data. The description of the corresponding data structures is given in **Listing 5**.

```
struct E6portq {
        struct E6portq * next;
        struct E6portq * prev;
        u16 E6reg_port;
        pid_t pid;
        int qlen;
        struct sk_buf * skbhead;
        struct sk_buf * skbtail;
}

static struct E6portq * E6inpq_head = NULL;
static struct E6portq * E6inpq_tail = NULL;

int E6_sendmsg( struct E6msg_buf __user *msg )
{
        int len;
        struct sk_buff *skb;
        struct E6hdr *E6h;
        struct ethhdr *eth;
        unsigned long flags;
        int rc=0;

        copy_from_user(km, msg, sizeof(struct E6msg_buf)));
        len=km->len;
        if( E6find_regport( E6src_port ) = = NULL )
        {
                rc = E6ERRUNREGPORT;
                return rc;
        }
        skb=alloc_skb(len+E6HEADSSPACE, GFP_KERNEL);
        skb->dev=E6_dev;
        skb->sk=NULL;

        /* Copy data */
        skb_reserve(skb, E6HEADSSPACE);
        copy_from_user( skb_put(skb,len), km->E6data, len );
        /* Build the E6P2 header. */
        skb_push(skb, sizeof(struct E6hdr));
        skb_reset_transport_header(skb);
        E6h = (struct E6hdr *)skb_transport_header(skb);
        E6h->E6dst_port = htons( km->E6dst_port );
        E6h->E6src_port = htons( km->E6src_port );
        skb_reset_network_header(skb);

        /* Build the E6Ethernet header */
        skb_push(skb, ETH_HLEN);
        skb_reset_mac_header(skb);
        eth = (struct ethhdr *)skb_mac_header(skb);
        skb->protocol = eth->h_proto = htons(ETH_P_E6);
        memcpy(eth->h_source, E6_myaddr, dev->addr_len);
        memcpy(eth->h_dest, km->E6dst_addr, dev->addr_len);

        /* Pass skb to the driver */
        local_irq_save(flags);
        netif_tx_lock(dev);
        rc=dev->hard_start_xmit(skb,dev);
        netif_tx_unlock(dev);
        local_irq_restore(flags);

        return rc;
}
```

Listing 5. Sending E6 messages via Ethernet driver.

Let us consider the sending of E6 messages in details (**Listing 5**). Routine E6_sendmsg copies the message header from the user address space (copy_from_user), checks whether the source port is registered, allocates skb, copies the data from the user address space, fills in E6h and eth headers and passes skb to the driver (hard_start_xmit).

Brief description of other routines is given in **Listing 6.**

```
int E6_getmsg( u16 __user *upn, struct E6msg_buf __user *msg )
        // finds registered port with the number pn=*upn
        // checks weather the port pn belongs to the current process
        // checks weather the skb queue to the port pn is empty
        // extracts the head element from skb queue
        // copies the message header and the data into the user data space
        // frees skb

int E6_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev)
        // extracts the destination port number dp from skb
        // finds registered port with the number dp; if unregistered, drops skb
        // checks the skb queue limit; if exceeded, drops skb
        // puts skb into the tail of port dp queue
        // checks the waiting flag and wakes up the waiting process

int E6_waitmsg(    u16 __user *upn )
        // checks weather the port pn=*upn is registered and belongs to the current process
        // checks the skb queue is empty; if empty, blocks the current processing

int E6_regport( u16 __user *upn )
        // checks weather the port pn=*upn is registered; if yes, returns an error
        // creates new E6portq record, fills it in and puts it to the registered ports queue

int E6_unregport( u16 __user *upn )
        // checks weather the port pn=*upn is registered; if no, returns an error
        // checks weather the port pn belongs to the current process; if no, returns an error
        // frees all the skb in the queue to the port pn (drops packets)
        // extracts E6portq record from the list and frees it

void E6udpmod_cleanup_module(void)
        // unregisters E6 packet type: dev_remove_pack(&E6_packet_type);
        // unregisters module system call handler: new_sys_E6call = NULL;
        // frees the ports list and the corresponding skb queues
```

Listing 6. Brief description of E6 module routines.

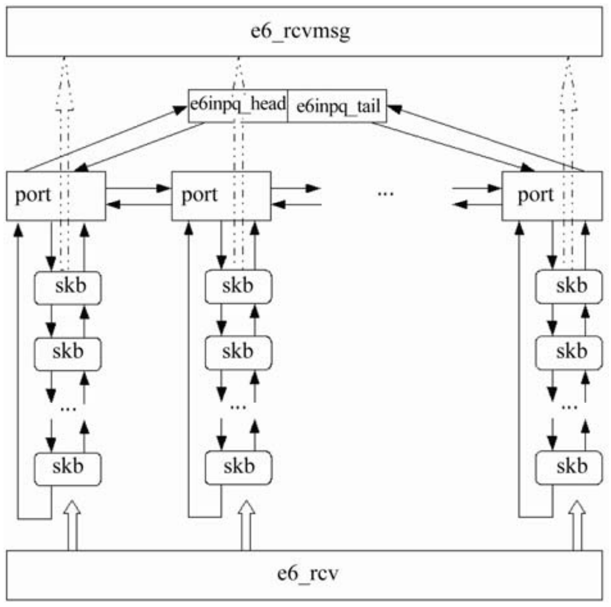

Figure 7. The basic data structures.

Note that the shown fragments of code are rather simplified for the brevity, numerous checks of the routines return codes and corresponding actions on errors were omitted.

6. Conclusions

It was implemented the first kernel level E6 protocol stacks on Linux, kernel version 2.6.22.9. The robustness of E6 network was confirmed by the successful running of E6talk application among other networking applications and protocols.

The datagram mode of the communication was implemented (UDP like). The implementation of the data segment mode with the guaranteed delivery of information (TCP like) on the base of Ethernet LLC2 is the direction for future work as well as the building gateways between E6 and TCP/IP networks which exist now in parallel.

REFERENCES

[1] P. P. Vorobiyenko, D. A. Zaitsev and O. L. Nechiporuk, “World-Wide Network Ethernet?” *Zviazok* (*Communications*), No. 5, 2007, pp. 14-19.

[2] P. P. Vorobiyenko, D. A. Zaitsev and K. D. Guliaiev, “Way of Data Transmission within Network with Substitution of Network and Transport Layers by Universal Technology of Layer,” Patent of Ukraine on Utility Model, No. 35773, 2008.

[3] K. D. Guliaiev, D. A. Zaitsev, D. A. Litvin and E. V. Radchenko, “Simulating E6 Protocol Networks Using

CPN Tools," Proceeding of International Conference on IT Promotion in Asia, 22-26 September 2008, Tashkent, pp. 203-208.

[4] J. Postel, "Transmission Control Protocol," Information Sciences Institute, University of Southern California, Los Angeles, No. RFC 793, 1981, p. 85.

[5] J. Postel, "User Datagram Protocol," Information Sciences Institute, University of Southern California, Los Angeles, No. RFC 768, 1980, p. 3.

[6] Carrier Sense Multiple Access with Collision Detection (CSMA/CD), "Access Method and Physical Layer Specifications—LAN/MAN Standards Committee of the IEEE Computer Society," *IEEE Std* 802.3-2005, *IEEE-SA Standards Board IP*, 9 June 2005, p. 417.