libPcap

Contents

1 Introduction

RAWSOCKETS have 2 disadvantages:

- they induce 1 system call by paquet
- change on the buffer size are global for the system

According to */linux/Documentation/networking/packet_mmap.txt*, PAQUET MMAP allocates a ring buffer on user space like MMAP do (version >=1.0 only). LIBPCAP uses natively the PAQUET MMAP allocation. However, according to `man 7 pcap-filter`, LIBPCAP also provides packet filtering handled by kernel. It is used by TCPDUMP for instance.

note: once packet read return, packet are no longer hosted in the ring buffer.

2 Python

try "pcap dpkt" on google.

DPKT allow to work offline on packet captured with TCPDUMP and saved as RAW:

```
# tcpdump -i eth6 -xx -vv -w out.bin
```

cf */svn/calice/online-sw/trunk/tests_eth_dif/rpcap.py*. **Note** that *rpcap.py* only works with messages length equal 16.

3 C

3.1 pcap_next_ex

Looking to the sources, it seems that packets are copied using `pcap_next_ex`: Yes, this means that, if the capture is using the ring buffer, using pcap_next() or pcap_next_ex() requires more copies than using pcap_loop() or pcap_dispatch(). If that bothers you, don't use pcap_next() or pcap_next_ex().

- file *libpcap-1.1.1/pcap.c*:

```
/*
 * Default one-shot callback; overridden for capture types where the
 * packet data cannot be guaranteed to be available after the callback
 * returns, so that a copy must be made.
 */
static void
```

```
pcap_oneshot(u_char *user, const struct pcap_pkthdr *h, const u_char *pkt)
{
struct oneshot_userdata *sp = (struct oneshot_userdata *)user;

*sp->hdr = *h;
*sp->pkt = pkt;
}

int
pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header,
    const u_char **pkt_data)
{
struct oneshot_userdata s;

s.hdr = &p->pcap_header;
s.pkt = pkt_data;
s.pd = p;

/* Saves a pointer to the packet headers */
*pkt_header= &p->pcap_header;

if (p->sf.rfile != NULL) {
int status;

/* We are on an offline capture */
status = pcap_offline_read(p, 1, pcap_oneshot, (u_char *)&s);

/*
 * Return codes for pcap_offline_read() are:
 *   -  0: EOF
 *   - -1: error
 *   - >1: OK
 * The first one ('0') conflicts with the return code of
 * 0 from pcap_read() meaning "no packets arrived before
 * the timeout expired", so we map it to -2 so you can
 * distinguish between an EOF from a savefile and a
 * "no packets arrived before the timeout expired, try
 * again" from a live capture.
 */
if (status == 0)
return (-2);
else
return (status);
}

/*
 * Return codes for pcap_read() are:
 *   -  0: timeout
 *   - -1: error
 *   - -2: loop was broken out of with pcap_breakloop()
 *   - >1: OK
 * The first one ('0') conflicts with the return code of 0 from
 * pcap_offline_read() meaning "end of file".
*/
return (p->read_op(p, 1, pcap_oneshot, (u_char *)&s));
}
```

- file *libpcap-1.1.1/pcap-linux*:

```
/*
 * Special one-shot callback, used for pcap_next() and pcap_next_ex(),
 * for Linux mmapped capture.
 *
 * The problem is that pcap_next() and pcap_next_ex() expect the packet
 * data handed to the callback to be valid after the callback returns,
 * but pcap_read_linux_mmap() has to release that packet as soon as
 * the callback returns (otherwise, the kernel thinks there's still
 * at least one unprocessed packet available in the ring, so a select()
 * will immediately return indicating that there's data to process), so,
 * in the callback, we have to make a copy of the packet.
 *
!!! HERE (read this) !!!
 * Yes, this means that, if the capture is using the ring buffer, using
 * pcap_next() or pcap_next_ex() requires more copies than using
 * pcap_loop() or pcap_dispatch().  If that bothers you, don't use
 * pcap_next() or pcap_next_ex().
!!! HERE !!!
 */
static void
pcap_oneshot_mmap(u_char *user, const struct pcap_pkthdr *h,
    const u_char *bytes)
{
struct oneshot_userdata *sp = (struct oneshot_userdata *)user;

*sp->hdr = *h;
memcpy(sp->pd->md.oneshot_buffer, bytes, h->caplen);
*sp->pkt = sp->pd->md.oneshot_buffer;
}
```

3.2 pcap_loop

cf:

- `$man pcap_loop`.
- Coding a Simple Packet Sniffer
- French version of the above document

3.3 Exemple

ping

```
#include <stdio.h>
#include <stdlib.h>    // exit

#include <pcap/pcap.h>
#include <arpa/inet.h> // htons, uint16_t

pcap_t* ethConnect(char* ethernetCard, char* filter)
{
  static char errbuff[PCAP_ERRBUF_SIZE];
  pcap_t* rc = (pcap_t*)0;
  struct bpf_program          pcap_filter;

  memset(& pcap_filter, 0, sizeof(pcap_filter));

  // create a live capture handle
```

```
  if ((rc = pcap_create(ethernetCard, errbuff)) == (pcap_t *)0){
    printf("pcap_create: %s\n", errbuff);
    goto error;
  }

  // activate a capture handle
  if (pcap_activate(rc) != 0){
    printf("pcap_activate failed.\n");
    goto error;
  }

  return rc;
 error:
  // close a capture device or savefile
  pcap_close(rc);

  // free a BPF program
  pcap_freecode(&pcap_filter);

  return (pcap_t*)0;
}

/*
  >>
  08 10 00 02
  00 00 00 00 00 0b

  30 0f 00 00 00 00
  30 0e 00 00 00 00
  30 00 00 00 00 00
  30 15 00 00 00 00
  30 10 00 00 00 00
  10 02 00 00 00 00
  10 00 00 00 00 00
  10 0a 00 00 00 00
  10 0e 00 00 00 00
  10 22 00 00 00 00
  10 24 00 00 00 00

  <<
  08 10 00 04
  00 00 00 00 00 0b
  30 0f 00 00 09 09   LDA_VERSION
  30 0e 00 00 00 01   LDA_REVISION
  30 00 00 00 00 00   LDA_ENABLES
  30 15 00 00 00 00   LDA_PKTGEN_TXCOUNT
  30 10 00 00 00 00   LDA_PKTGEN_CONTROL
  10 02 00 00 03 ff   DIF_LINK_RX_EN
  10 00 00 00 03 ff   DIF_LINK_TX_EN
  10 0a 00 00 00 00   DIF_LINK_STATUS1
  10 0e 00 00 00 00   DIF_LINK_NO_SIGNAL
  10 22 00 00 00 00   DIF_LINK_LOCKED
  10 24 00 00 00 01   DIF_LINK_DCM
  */
int buildPing(uint16_t** rc)
{
  static uint16_t ldapkt[100];
  int offs = 0;
```

```
  int i;

  // lda mac
  ldapkt[offs++] = htons(0x5e70);
  ldapkt[offs++] = htons(0x0cd2);
  ldapkt[offs++] = htons(0x77e8);

  // api mac
  ldapkt[offs++] = htons(0x0800);
  ldapkt[offs++] = htons(0x27a8);
  ldapkt[offs++] = htons(0xf0e8);

  ldapkt[offs++] = htons(0x0810); // ether type
  ldapkt[offs++] = htons(0x0002); // subsystem + optype
  ldapkt[offs++] = htons(0x0000); // modifier
  ldapkt[offs++] = htons(0x0000); // packet id
  ldapkt[offs++] = htons(0x000b); // data length

  ldapkt[offs++] = htons(0x300f); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);
  ldapkt[offs++] = htons(0x300e); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);
  ldapkt[offs++] = htons(0x3000); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);
  ldapkt[offs++] = htons(0x3015); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);
  ldapkt[offs++] = htons(0x3010); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);
  ldapkt[offs++] = htons(0x1002); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);
  ldapkt[offs++] = htons(0x1000); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);
  ldapkt[offs++] = htons(0x100a); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);
  ldapkt[offs++] = htons(0x100e); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);
  ldapkt[offs++] = htons(0x1022); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);
  ldapkt[offs++] = htons(0x1024); ldapkt[offs++] = htons(0x0000); ldapkt[offs++] = htons(0x0000);

  //printf("Packet is %i bytes:\n", offs*sizeof(uint16_t));
  for (i=0; i< offs; ++i) {
    if (i%8==0) printf("\n");
    printf ("%02x %02x ", ldapkt[i]>>8, ldapkt[i]&0xff);
  }
  printf("\n");

  *rc = ldapkt;
  return offs;
}

/*
//CALDIF::PktAccessor_LDA_pkt ldapkt(66);
//ldapkt.set_pdu_nbytes(0);  int i;

//uint8_t const ldaAdd[ETHER_ADDR_LEN]={0x5e, 0x70, 0x0c, 0xd2, 0x77, 0xe8};
////uint8_t const apiAdd[ETHER_ADDR_LEN]={0x00, 0x22, 0x19, 0x00, 0x3d, 0x8b};
//uint8_t const apiAdd[ETHER_ADDR_LEN]={0x08, 0x00, 0x27, 0xA8, 0xF0, 0xE8};
//struct ether_addr lda_mac;

//LLR_CALICE_TESTS::LDA_interface lldapkt[i]da("eth2", lda_mac);
//lda.set_debug();

//lda.flush_recv_queue();
//lda.set_intersend_pause(10);
//lda.send_tcpdump_magic(0x22, 'b');

//ldapkt.set_ether_dhost(ldaAdd);
```

```
//ldapkt.set_ether_shost(apiAdd);

//ldapkt.set_ether_type(0x0810);
//ldapkt.set_LDA_subsystem(0x00);
//ldapkt.set_LDA_optype(0x02);
//ldapkt.set_LDA_Modifier(0x0000);
//ldapkt.set_LDA_PktID(0x0000);
//ldapkt.set_LDA_DataLength(0x000b);

//uint16_t * ldacontents = (uint16_t*) ldapkt.get_pdu();
//lda.send_verbatim(ldapkt);
*/
int main()
{
  pcap_t *pcap = NULL;
  int rc = 0;
  char* filter = NULL;
  uint16_t* ldapkt = NULL;
  int offs = 0;

  offs = buildPing(&ldapkt);
  pcap = ethConnect("eth2", filter);

  // transmit a packetpkt
  if ((rc = pcap_inject(pcap, ldapkt, offs*2)) < 0){
    pcap_perror(pcap, "pcap_inject: ");
    goto error;
  }

  //printf("%i bytes injected\n", rc);

  // close a capture device or savefile
  pcap_close(pcap);

  exit(EXIT_SUCCESS);
 error:
  exit(EXIT_FAILURE);
}
```

listen

```
#include <stdio.h>
#include <string.h> // memset
#include <stdlib.h> // exit

#include <pcap/pcap.h>
#include <sys/poll.h>

#ifndef FALSE
#define FALSE 0
#endif

#ifndef TRUE
#define TRUE 1
#endif


/*
//LLR_CALICE_TESTS::LDA_interface lda("eth2", lda_mac);
```

```
//lda.set_debug();
//lda.flush_recv_queue();
*/
pcap_t* ethConnect(char* ethernetCard, char* filter)
{
  static char errbuff[PCAP_ERRBUF_SIZE];
  pcap_t* rc = (pcap_t*)0;
  struct bpf_program          pcap_filter;

  memset(& pcap_filter, 0, sizeof(pcap_filter));

  // create a live capture handle
  if ((rc = pcap_create(ethernetCard, errbuff)) == (pcap_t *)0){
    printf("pcap_create: %s\n", errbuff);
    goto error;
  }

  // set the snapshot length for a not-yet-activated capture handle
  if (pcap_set_snaplen(rc, 16*1024) != 0){
    printf("pcap_set_snaple failed.\n");
    goto error;
  }

  // set the buffer size for a not-yet-activated capture handle
  if (pcap_set_buffer_size(rc, 8UL*1024*1024) != 0){
    printf("pcap_set_buffer_size failed.\n");
    goto error;
  }

  /* We need to set a timeout > 0 so that pcap_setnonblock() below
     takes effect. If we keep the default value (0), then
     pcap_setnonblock() will have no effect, and any read operation
     will be infinitly blocking... which is the OPPOSITE from what
     we want. With a timeout > 0, then pcap_setnonblock() will
     effectively and correctly configure the interface as
     non-blocking. */

  // set the read timeout for a not-yet-activated capture handle
  if (pcap_set_timeout(rc, 10) != 0){
    printf("pcap_set_timeout failed.\n");
    goto error;
  }

  // activate a capture handle
  if (pcap_activate(rc) != 0){
    printf("pcap_activate failed.\n");
    goto error;
  }

  // compile a filter expression
  if (pcap_compile(rc, &pcap_filter, filter, 1, 0) != 0){
    pcap_perror(rc, "pcap_compile: ");
    goto error;
  }

  // set the filter
  if (pcap_setfilter(rc, &pcap_filter) != 0){
    pcap_perror(rc, "pcap_setfilter: ");
```

```
    goto error;
  }

  // set or get the state of non-blocking mode on a capture device
  if (pcap_setnonblock(rc, 1, errbuff) != 0){
    printf("pcap_setnonblock: %s\n", errbuff);
    goto error;
  }

  return rc;
 error:
  // close a capture device or savefile
  pcap_close(rc);

  // free a BPF program
  pcap_freecode(&pcap_filter);

  return (pcap_t*)0;
}

/*
  CALDIF::PktAccessor_LDA_pkt const* ldapkt = NULL;
  try
  {
  ldapkt = & lda.recv(100000000);
  }
  catch (LLR_CALICE_TESTS::LDAError & ex)
  {
  LLR_LOGGER_ERROR("Got error while waiting for pkt: "
  << ex.what());
  break;
  }

  std::stringstream sstrm;
  CALDIF::PktPrinter_Human.print(sstrm << "Received packet ",
  *ldapkt) << std::endl;
  CALDIF::hexdump(sstrm, ldapkt->get_sdu(), ldapkt->get_sdu_nbytes()
  - sizeof(struct ETH_word_CRC));
  LLR_LOGGER_DEBUG(sstrm.str());
*/

int
waitForPacket(int sd, unsigned timeout_ms)
{
  struct pollfd pfd;
  pfd.fd      = sd;
  pfd.events  = POLLIN|POLLRDNORM|POLLERR;
  pfd.revents = 0;
  return (1 == ::poll(&pfd, 1, timeout_ms));
}

int
ethRecv (pcap_t* pcap,
 struct pcap_pkthdr **pkt_header, const unsigned char **pkt_data,
 int timeout_ms)
{
  static char errbuff[PCAP_ERRBUF_SIZE];
  int rc = 0;
```

```
  int sd = -1;

  // get the file descriptor for a live capture
  if ((sd = pcap_fileno(pcap)) < 0){
    printf("pcap_fileno: %s\n", errbuff);
    goto error;
  }

  /* Try first to read from user memory */

  // read the next packet from a pcap_t
  if ((rc = pcap_next_ex(pcap, pkt_header, pkt_data)) < 0){
    printf("pcap_next_ex: %s\n", errbuff);
    goto error;
  }

  if (timeout_ms > 0
      && rc == 0 /* No packet immediatly available... */)
    {
      /* No packet => wait for one to arrive (negative timeout =
 infinite) */
      waitForPacket(sd, timeout_ms); /* Ignore return value to
 prevent famine */

      /* Second chance after poll()... */

      // read the next packet from a pcap_t
      if ((rc = pcap_next_ex(pcap, pkt_header, pkt_data)) < 0){
printf("pcap_next_ex: %s\n", errbuff);
goto error;
      }
    }

  return rc;
 error:
  return -1;
}

int main()
{
  unsigned int i;
  pcap_t               *pcap;
  struct pcap_pkthdr   *pkt_header;
  const unsigned char  *pkt_data;
  char* filter = \
    "( " \
    "    ether proto 0x0809 " \
    " or ether proto 0x0810 " \
    " or ether proto 0x0811" \
    ") " \
    "and ether src 5e:70:0c:d2:77:e8";

  pcap = ethConnect("eth2", filter);

  while (ethRecv(pcap, &pkt_header, &pkt_data, 10000000) > 0) {

    for (i=0; i< pkt_header->caplen; ++i) {
      if (i%16==0) printf("\n");
```

```
      printf ("%02x ", pkt_data[i]);
    }
    printf("\n");
  }

  printf("Timeout waiting for packet\n");

  exit(EXIT_SUCCESS);
}
```