#include "ProtoSocket.h"

#include <cstdio>

int main()

{

printf("Protocol process starting...\n");

CProtoSocket\* protoSocket = CProtoSocket::instance();

protoSocket->run();

printf("Protocol process exiting...\n");

return 0;

}

#ifndef ICMP\_H\_

#define ICMP\_H\_

typedef struct sniff\_icmp {

uint8\_t icmp\_type; // type of message

uint8\_t icmp\_code; // type sub code

uint16\_t icmp\_sum; // one complement check sum of struct

union {

uint8\_t ih\_pptr; // parameter problem pointer

struct in\_addr ih\_gwaddr; // Gateway Internet Address

struct ih\_idseque {

uint16\_t icd\_id; // identifier

uint16\_t icd\_seq; // sequence number

} ih\_idseque ;

uint32\_t ih\_void;

} icmp\_hun ;

#define icmp\_pptr\_t icmp\_hun.ih\_pptr

#define icmp\_gwaddr\_t icmp\_hun.ih\_gwaddr

#define icmp\_id\_t icmp\_hun.ih\_idseque.icd\_id

#define icmp\_seq\_t icmp\_hun.ih\_idseque.icd\_seq

#define icmp\_void\_t icmp\_hun.ih\_void

union {

struct id\_ts {

uint32\_t its\_otime; // Originate timestamp

uint32\_t its\_rtime; // Receive timestamp

uint32\_t its\_ttime; // Transmit timestamp

} id\_ts;

struct id\_ip {

struct sniff\_ip idi\_ip;

/\*options and then 64bits of data\*/

} id\_ip;

uint32\_t id\_mask;

uint8\_t id\_data[1];

} icmp\_dun ;

#define icmp\_otime\_t icmp\_dun.id\_ts.its\_otime

#define icmp\_rtime\_t icmp\_dun.id\_ts.its\_rtime

#define icmp\_ttime\_t icmp\_dun.id\_ts.its\_ttime

#define icmp\_ip\_t icmp\_dun.id\_ip.idi\_ip

#define icmp\_mask\_t icmp\_dun.id\_mask

#define icmp\_data\_t icmp\_dun.id\_data

} icmphdr\_t ;

#endif // ICMP\_H\_

#pragma once

#include "SharedBlock.h"

#include "InetSocket.h"

#include "packet.h"

#include <map>

#include <set>

#include <string>

typedef std::map<std::string, InetConnSock\*> ConnPMap; // connections, <localAddr.localPort-peerAddr.peerPort, InetConnSock\*>

// actually, value is a pointer to TCP member \_connPool's element

class CProtoSocket

{

public:

static CProtoSocket\* instance()

{

static CProtoSocket inst;

return &inst;

}

~CProtoSocket();

void run();

/\*

\* Received packet from protocol

\* \*/

void received(const packet\_t \*pkt);

/\*

\* Called by TCP when a new connection @ics is arrived

\*

\* @name Connection name

\* @ics The connection

\*

\* To be honest, this method can be avoided

\* \*/

void connectFinished(std::string name, InetConnSock \*ics);

uint32\_t selectFD();

/\*

\* Accepted an established connection from TCP

\*

\* @name Connection name

\* @ics The connection

\* \*/

void accepted(std::string name, InetConnSock \*ics);

void bytesAvailable(InetConnSock \*ics);

void closed(std::string name);

private:

//

CProtoSocket();

CProtoSocket(const CProtoSocket&);

CProtoSocket & operator=(const CProtoSocket&);

/\*

\* Create and detach shared memory

\* \*/

void createSharedMem();

void destroySharedMem();

void handleSockRequest();

void handleCreate(SockPacket \*sockPkt);

void handleBind(SockPacket \*sockPkt);

void handleSendTo(SockPacket \*sockPkt);

void handleRecvFrom(SockPacket \*sockPkt);

void handleSend(SockPacket \*sockPkt);

void handleRecv(SockPacket \*sockPkt);

void handleClose(SockPacket \*sockPkt);

void handleListen(SockPacket \*sockPkt);

void handleConnect(SockPacket \*sockPkt);

void handleAccept(SockPacket \*sockPkt);

/\*

\* Random select an unused port

\* \*/

uint16\_t selectPort();

void setLocalAddr(InetSock \* sk);

/\*

\* Notify socket a given signal

\*

\* @success An int value when 1 for success, 0 for failed

\* @pid The process id

\* @signo The signal number

\* @funcName The calling function name

\* \*/

void afterHandle(int success, int pid, int signo, const char \* const funcName);

void afterHandle(int pid, int signo, const char \* const funcName);

std::set<uint16\_t> \_pendingAccept;

ConnPMap \_connPPool; // connection pointers map

std::map<int, InetSock> \_sockPool; // created sockets, <sockfd, InetSock>

std::set<InetSock \*> \_pendingSocks; // pending recvfrom sockets

int \_shmid; // shared memory identifier

SharedBlock \*\_pBlock;// shared block

};

#pragma once

#include "tcp.h"

#include "ProtoSocket.h"

#include "InetSocket.h"

#include "Network.h"

#include "BaseIO.h"

#include <map>

#include <string>

typedef std::map<std::string, InetConnSock> ConnMap; // connections, <localAddr.localPort-peerAddr.peerPort, InetConnSock>

typedef std::map<uint16\_t, InetSock\*> InetSockMap; // listen sockets, <port, InetSock\*>

// actually, value is a pointer to CProtoSocket member \_sockPool's

// element

class CNetwork;

class CProtoSocket;

class CTCP : public CBaseIO

{

public:

static CTCP \* instance()

{

static CTCP inst;

return &inst;

}

~CTCP();

void init();

int send(packet\_t \*pkt);

int received(packet\_t \*pkt);

void connect(InetSock \*sk);

void listen(InetSock \*sk);

/\*

\* Close a connection

\*

\* @name The connection name, can be contructed by keyOf()

\* \*/

void close(std::string name);

/\*

\* Return the key(or name) of a connection.

\* \*/

static std::string keyOf(InetConnSock \*ics);

static std::string keyOf(struct in\_addr localAddr, uint16\_t localPort, struct in\_addr peerAddr, uint16\_t peerPort);

private:

CTCP(const CTCP&);

CTCP & operator=(const CTCP&);

CTCP() { }

InetConnSock \* newConnection(InetConnSock \*ics);

void doSend(InetConnSock \*ics);

void \_\_doSend(packet\_t \*packet, InetConnSock \*ics, uint8\_t flags, uint8\_t \*buf, uint32\_t size);

/\*

\* Send a tcp segment without payload

\*

\* @packet The allocated packet

\* @ics The connection

\* @flags The tcp header flags

\* \*/

void sendNoData(packet\_t \*packet, InetConnSock \*ics, uint8\_t flags);

void recvEstablished(InetConnSock \*ics, packet\_t \*packet, tcphdr\_t \*tcphdr);

void recvStateProcess(InetConnSock \*ics, packet\_t \*packet, tcphdr\_t \*tcphdr);

void recvListen(InetSock \*ics, packet\_t \*packet, tcphdr\_t \*tcphdr);

InetSockMap \_listenPool;

ConnMap \_connPool;

CProtoSocket \*\_protoSock;

CNetwork \*\_network;

};

#pragma once

#include <arpa/inet.h>

#include <netinet/ip.h>

#include <memory>

#include "SharedBlock.h"

class CSocket

{

public:

CSocket();

~CSocket();

int socket(int family, int type, int protocol);

/\*

\* Bind socket with address @sockaddr

\* \*/

int bind(const struct sockaddr \*sockaddr, socklen\_t addrlen);

/\*

\* Connect to address @sockaddr

\* \*/

int connect(const struct sockaddr \*sockaddr, socklen\_t addrlen);

/\*

\* Send to connected peer address

\*

\* @buf The data to send

\* @len The data length in bytes to send

\* \*/

int send(const char \* buf, size\_t len, int flags);

/\*

\* Receive data from connected peer address, default timeout is 3 seconds.

\*

\* @buf The buffer used to stored received data

\* @len Maximum bytes to receive

\* \*/

int recv(char \* buf, size\_t len, int flags);

/\*

\* @return The number of bytes sent. On error, -1 is returned

\* \*/

int sendto(const char\* buf, size\_t len, int flags,

const struct sockaddr\* dstAddr, socklen\_t addrlen);

/\*

\* @return The number of bytes received, or -1 if an error occurred.

\* \*/

int recvfrom(char\* buf, size\_t len, int flags,

struct sockaddr\* srcAddr, socklen\_t\* addrlen);

/\*

\* Set this socket to listen mode

\*

\* @backlog The maximum client might in queue

\* @return On success, zero is returned. On error, -1 is returned

\* \*/

int listen(int backlog);

std::unique\_ptr<CSocket> accept(struct sockaddr \* sockaddr, socklen\_t \* addrlen);

int shutdown();

int close();

/\*

\* Return file descriptor of this socket

\* \*/

int getFD()

{

return \_sock.sockfd;

}

//

private:

//

CSocket(const CSocket&); // prevent copy

CSocket & operator= (const CSocket&);// prevent assign

/\*

\* Do socket create

\* \*/

int init(int family, int type, int protocol);

/\*

\* Attach and detach shared memory

\* \*/

void attachSharedMem();

void detachSharedMem();

/\*

\* Wait protocols stack reply

\*

\* @return Theorically, on success, 1 is returned, otherwise return 0 !!!

\* \*/

int waitForSuccess(int signo);

SockDataHdr makeDataHeader(size\_t len, int flag);

int \_shmid; // shared memory identifier

SharedBlock \*\_pBlock;// shared block

int \_protoPid; // protocol process id

Sock \_sock;

//int \_socketId; // might use process id

//int \_family;

//int \_type;

//int \_protocol;

};

#ifndef TCP\_H\_

#define TCP\_H\_

#include <sys/types.h>

#define SIZE\_TCP 20

/\*

\* TCP Maximum Segment Size (MTU - SIZE\_TCP - SIZE\_IP)

\* \*/

const int cTCPMSS = 1456;

/\* TCP header \*/

typedef u\_int tcp\_seq;

typedef struct sniff\_tcp {

u\_short th\_sport; /\* source port \*/

u\_short th\_dport; /\* destination port \*/

tcp\_seq th\_seq; /\* sequence number \*/

tcp\_seq th\_ack; /\* acknowledgement number \*/

u\_char th\_offx2; /\* data offset, rsvd \*/

#define TH\_OFF(th) (((th)->th\_offx2 & 0xf0) >> 4)

u\_char th\_flags;

#define TH\_FIN 0x01

#define TH\_SYN 0x02

#define TH\_RST 0x04

#define TH\_PUSH 0x08

#define TH\_ACK 0x10

#define TH\_URG 0x20

#define TH\_ECE 0x40

#define TH\_CWR 0x80

#define TH\_FLAGS (TH\_FIN|TH\_SYN|TH\_RST|TH\_ACK|TH\_URG|TH\_ECE|TH\_CWR)

u\_short th\_win; /\* window \*/

u\_short th\_sum; /\* checksum \*/

u\_short th\_urp; /\* urgent pointer \*/

} tcphdr\_t ;

/\*

\* TCP control buffer

\* \*/

typedef struct tagTCPCB {

// ...

u\_int seq; // start sequence of buffer

u\_int endSeq; // end sequence of buffer

u\_int ack; // acked number

} TCPCB;

#define TCP\_PKT\_CB(\_\_packet) ((TCPCB \*)&(\_\_packet)->cb[0])

#endif // TCP\_H\_

#ifndef ETHER\_H\_

#define ETHER\_H\_

#include <netinet/ether.h>

/\* ethernet headers are always exactly 14 bytes [1] \*/

#define SIZE\_ETHERNET 14

#define SIZE\_ETHER\_SUM 4

/\* Ethernet addresses are 6 bytes \*/

#ifndef ETHER\_ADDR\_LEN

#define ETHER\_ADDR\_LEN 6

#endif

/\*MAC address in ASCII format length\*/

#define MAC\_ASCII\_LEN 18

/\*

\* Ethernet Type Defines, see /usr/include/net/ethernet.h

\* \*/

/\*#define ETHER\_T\_IPv4 0x0800 // Internet Protocol Version 4\*/

/\*#define ETHER\_T\_IPv6 0x86DD // Internet Protocol Version 6\*/

/\*#define ETHER\_T\_ARP 0x0806 // Address Resolution Protocol \*/

/\*#define ETHER\_T\_RARP 0x8035 // Reverse Address Resolution Protocol \*/

/\*#define ETHER\_T\_ETHERTALK 0x809B // AppleTalk over Ethernet\*/

/\*#define ETHER\_T\_PPP 0x880B // Point-to-Point Protocol\*/

/\*#define ETHER\_T\_PPPoEDS 0x8863 // PPPoE Discovery Stage\*/

/\*#define ETHER\_T\_PPPoESS 0x8864 // PPPoE Session Stage\*/

/\*#define ETHER\_T\_SNMP 0x814C // Simple Network Management Protocol\*/

/\* Ethernet header \*/

typedef struct sniff\_ethernet {

u\_char ether\_dhost[ETHER\_ADDR\_LEN]; /\* destination host address \*/

u\_char ether\_shost[ETHER\_ADDR\_LEN]; /\* source host address \*/

u\_short ether\_type; /\* IP? ARP? RARP? etc \*/

} ethernethdr\_t ;

#endif // ETHER\_H\_

#pragma once

#include "udp.h"

#include "BaseIO.h"

class CUDP : public CBaseIO

{

public:

static CUDP \* instance()

{

static CUDP inst;

return &inst;

}

//

void init();

/\*

\* Send pkt out.

\*

\* @pkt The packet to send

\* \*/

int send(packet\_t \*pkt);

/\*

\* Receive a packet pkt.

\*

\* @pkt The packet received

\* \*/

int received(packet\_t \*pkt);

private:

CUDP()

{

}

CUDP(const CUDP&);

CUDP & operator= (const CUDP&);

};

#ifndef IP\_H\_

#define IP\_H\_

#include <netinet/in.h>

#include <netinet/ip.h>

/\* to remove option, just set to 0 \*/

#define SIZE\_OPTION 4

#define SIZE\_IP (SIZE\_OPTION+20)

#define SIZE\_IP\_HL (SIZE\_IP/4)

const int cIPOptionValue = 0xFF020000;

/\* IP header \*/

struct sniff\_ip {

u\_char ip\_vhl; /\* version << 4 | header length >> 2 \*/

u\_char ip\_tos; /\* type of service \*/

u\_short ip\_len; /\* total length \*/

u\_short ip\_id; /\* identification \*/

u\_short ip\_off; /\* fragment offset field \*/

#define IP\_RF 0x8000 /\* reserved fragment flag \*/

#define IP\_DF 0x4000 /\* dont fragment flag \*/

#define IP\_MF 0x2000 /\* more fragments flag \*/

#define IP\_OFFMASK 0x1fff /\* mask for fragmenting bits \*/

u\_char ip\_ttl; /\* time to live \*/

u\_char ip\_p; /\* protocol \*/

u\_short ip\_sum; /\* checksum \*/

struct in\_addr ip\_src,

ip\_dst; /\* source and dest address \*/

u\_int ip\_opt; /\* option value \*/

bool isFragment() {

return (ip\_off & htons(IP\_MF | IP\_OFFMASK)) != 0;

}

};

typedef struct sniff\_ip iphdr\_t;

#define IP\_HL(ip) (((ip)->ip\_vhl) & 0x0f)

#define IP\_V(ip) (((ip)->ip\_vhl) >> 4)

#endif // IP\_H\_

#pragma once

//#ifndef NEIGHBOR\_H\_

//#define NEIGHBOR\_H\_

#include "Link.h"

#include "arp.h"

#include "packet.h"

class CARP;

class CLink;

/\*

\* CNeighbor - neighbor subsystem

\* \*/

class CNeighbor

{

public:

//

static CNeighbor \* instance()

{

static CNeighbor inst;

return &inst;

}

void init();

void send(packet\_t \*packet);

void received(packet\_t \*packet);

private:

bool \_isInited;

CLink \*\_link;

CARP \*\_arp;

CNeighbor() : \_isInited(false), \_arp(nullptr)

{

}

CNeighbor(const CNeighbor &);

CNeighbor & operator= (const CNeighbor &);

};

//#endif // NEIGHBOR\_H\_

#pragma once

#include <arpa/inet.h>

const char \* const cKeyPath = "/home/bdg/.netsimul"; // this file must exist

const int cKeyID = 17;

const unsigned int cSHMSize = 8192;

const unsigned int cSHMBufSize = 4096;

const unsigned int cSHMDataSize = 4092;

/\*

\* struct sharedblock - a data structure between ProtocolSocket and AppSocket

\*

\* @buf1 - ProtocolSocket buffer

\* @buf2 - AppSocket buffer

\* \*/

typedef struct sharedblock {

char buf1[cSHMBufSize]; // address is &sharedblock

char buf2[cSHMBufSize]; // address is buf1 + 4096

} SharedBlock;

typedef enum sockpktype {

SOCK\_CREATE = 1,

SOCK\_BIND,

SOCK\_LISTEN,

SOCK\_CONNECT,

SOCK\_ACCEPT,

SOCK\_SEND,

SOCK\_SENDTO,

SOCK\_RECV,

SOCK\_RECVFROM,

SOCK\_CLOSE,

} SockPktType;

typedef enum socketstate {

SS\_FREE = 0,

SS\_UNCONNECTED,

SS\_CONNECTED,

SS\_CONNECTING,

SS\_DISCONNECTING

} SocketState;

/\*

\* struct sockpacket - this data structure will save to buf1 or buf2 after conversion

\* \*/

typedef struct sockpacket {

SockPktType type; // this field must be the first member

char data[cSHMBufSize - 4];

} SockPacket;

/////////////// start of sockpacket.data structures ///////////////

typedef struct tagSock {

int sockfd; // socket file descriptor

int pid; // process id

int family;

int type;

int protocol;

struct in\_addr addr;// socket bind address

uint16\_t port; // socket bind port

struct in\_addr peerAddr;// peer socket address

uint16\_t peerPort; // peer socket port

SocketState state; // connection state

tagSock()

{

sockfd = pid = family = type = protocol = 0;

port = peerPort = 0;

addr.s\_addr = peerAddr.s\_addr = 0;

state = SS\_UNCONNECTED;

}

} Sock;

typedef struct tagSockDataHdr {

int sockfd;

struct sockaddr\_in srcAddr; // source address

struct sockaddr\_in dstAddr; // destination address

int flag; // flag

int len; // data length

} SockDataHdr;

/////////////// end of sockpacket.data structures ///////////////

#pragma once

//#ifndef PACKET\_H\_

//#define PACKET\_H\_

#include <arpa/inet.h>

#include <netinet/ether.h>

#include <cstring>

#include <cstdio> // use printf to see how many copy used

#include "ip\_arp.h"

struct packet\_cb {

// ...

uint16\_t offset;

};

typedef packet\_cb packet\_cb\_t;

typedef packet\_cb PacketCB;

/\*

\* Core structure used for networking, referece to Linux struct sk\_buff design.

\* \*/

struct inject\_packet {

u\_char \*buf; // packet data send buffer

size\_t size; // size of data bufer

struct in\_addr saddr, // source ip address

daddr; // destination ip address

struct ether\_addr sha, // source hardware address

dha; // destination hardware address

uint16\_t sport, // source port

dport; // destination port

u\_char oper; // operation code

unsigned char proto; // protocol

#define INJECT\_OP\_TCP(p) (p->oper == 't' || p->oper == 'T')

#define INJECT\_OP\_UDP(p) (p->oper == 'u' || p->oper == 'U')

uint16\_t ept; // ethernet packet type

//ARPHdr arphdr; // arp header

//const u\_char \*rcvbuf;// packet data receive buffer

char cb[48]; // control buffer for every process handler

bool allocated;// True if this struct allocated by heap memory

unsigned int len; // Length of the actual data

unsigned char \*tail; // Data tail

unsigned char \*end; // Buffer end

unsigned char \*head, // Buffer head

\*data; // Data pointer

inject\_packet() : size(0), len(0), allocated(false)

{

buf = head = tail = data = tail = nullptr;

//rcvbuf = nullptr;

}

inject\_packet(unsigned int size) : size(size), len(0), allocated(true)

{

buf = new unsigned char[size]{0};

head = data = tail = buf;

end = buf + size;

}

/\*

\* Copy constructor.

\*

\* So, remember to use reference& in any where you can.

\* \*/

inject\_packet (const inject\_packet &cp)

{

buf = new unsigned char[cp.size]{0};

head = data = tail = buf;

end = buf + cp.size;

data = head + (cp.data - cp.head);

tail = head + (cp.tail - cp.head);

len = cp.len;

size = cp.size;

saddr = cp.saddr;

daddr = cp.daddr;

sha = cp.sha;

dha = cp.dha;

sport = cp.sport;

dport = cp.dport;

oper = cp.oper;

proto = cp.proto;

ept = cp.ept;

//arphdr = cp.arphdr;

allocated = true;

memmove(buf, cp.buf, cp.size);

printf("\*\*Copied packet\*\*\n");

}

void copyMetadata(const inject\_packet &cp)

{

saddr = cp.saddr;

daddr = cp.daddr;

sha = cp.sha;

dha = cp.dha;

sport = cp.sport;

dport = cp.dport;

oper = cp.oper;

proto = cp.proto;

ept = cp.ept;

}

~inject\_packet()

{

if (allocated && buf != nullptr) {

delete [] buf;

buf = nullptr;

head = data = tail = end = buf;

size = len = 0;

}

}

/\*

\* Reserve space of headroom.

\*

\* Increase the headroom of an empty &sk\_buff by reducing the tail

\* room. This is only allowed for an empty buffer.

\* \*/

void reserve(unsigned int length)

{

data += length;

tail += length;

}

/\*

\* Add data length to a buffer.

\*

\* Extends the used data area of the buffer.

\* \*/

void put(unsigned int length)

{

tail += length;

len += length;

}

/\*

\* Add data to the start of a buffer.

\*

\* Extends the used data area of the buffer at the buffer start.

\* \*/

void push(unsigned int length)

{

data -= length;

len += length;

}

/\*

\* Remove data from the start of a buffer.

\*

\* \*/

void pull(unsigned int length)

{

data += length;

len -= length;

}

/\*

\* Trim data from the end of this buffer, Fri 23 Mar 2018 19:28:04

\* \*/

void trim(unsigned int length)

{

tail -= length;

len -= length;

}

/\*

\* Reset data pointer to head, and recalculate length.

\* \*/

void resetData()

{

data = head;

len = tail - data;

}

/\*

\* @return The packet control buffer pointer

\* \*/

packet\_cb\_t \* getPacketCB() const

{

return (packet\_cb\_t \*)cb;

}

bool isFull() const

{

return len == size;

}

bool empty() const

{

return len == 0;

}

};

typedef struct inject\_packet packet\_t;

typedef struct inject\_packet Packet;

//#endif // PACKET\_H\_

#pragma once

#include "SharedBlock.h"

#include "packet.h"

#include <string>

#include <list>

#include <memory>

/\*

\* enum use capital, remember this rule

\* \*/

typedef enum inetsockstate {

CLOSED = 0,

SYN\_SENT,

ESTABLISHED,

FIN\_WAIT\_1,

FIN\_WAIT\_2,

TIME\_WAIT,

LISTEN,

SYN\_RCVD,

CLOSE\_WAIT,

LAST\_ACK,

} InetSockState;

/\*

\* struct InetSock - Internet socket

\* \*/

typedef struct tagInetSock

{

Sock \_sock;

#define sk\_sockfd \_sock.sockfd

#define sk\_pid \_sock.pid

#define sk\_family \_sock.family

#define sk\_type \_sock.type

#define sk\_protocol \_sock.protocol

#define sk\_addr \_sock.addr

#define sk\_port \_sock.port

#define sk\_peerAddr \_sock.peerAddr

#define sk\_peerPort \_sock.peerPort

InetSockState sk\_state;

int backlog;

} InetSock;

typedef std::list<std::shared\_ptr<packet\_t>> PacketQueue;

typedef struct tagICSWindow

{

uint16\_t size; // window size

uint32\_t lastSeq;

uint32\_t lastAck;

uint32\_t nextSeq;

uint32\_t nextAck;

uint32\_t lower; // window lower bound

uint32\_t upper; // window upper bound

} ICSWindow;

/\*

\* struct InetConnSock - Internet connection based socket

\* \*/

typedef struct tagInetConnSock

{

InetSock \_inetSock;

#define ics\_sockfd \_inetSock.sk\_sockfd

#define ics\_pid \_inetSock.sk\_pid

#define ics\_family \_inetSock.sk\_family

#define ics\_type \_inetSock.sk\_type

#define ics\_protocol \_inetSock.sk\_protocol

#define ics\_addr \_inetSock.sk\_addr

#define ics\_port \_inetSock.sk\_port

#define ics\_peerAddr \_inetSock.sk\_peerAddr

#define ics\_peerPort \_inetSock.sk\_peerPort

#define ics\_state \_inetSock.sk\_state

ICSWindow sendWin,

recvWin;

PacketQueue sendQueue,

recvQueue;

tagInetConnSock()

{

memset(&sendWin, 0 , sizeof(ICSWindow));

memset(&recvWin, 0 , sizeof(ICSWindow));

}

} InetConnSock;

#ifndef NETSIMUL\_H\_

#define NETSIMUL\_H\_

char \_\_ch;

#define CLEAR() \

while((\_\_ch = getchar()) != '\n' && \_\_ch != EOF)

/\* default snap length (maximum bytes per packet to capture) \*/

#define SNAP\_LEN 1518

#define SIZE\_TOK\_BUF 256

typedef struct tok {

int v; // value

const char \* s; // string

} tok\_t ;

#endif // NETSIMUL\_H\_

#pragma once

//#ifndef ARP\_H\_

//#define ARP\_H\_

#include "packet.h"

#include "Link.h"

#include <sys/types.h>

#include <map>

#include <list>

#include <string>

using std::map;

using std::list;

const uint16\_t cARPMaxTTL = 500;

/\*

\* ARPPacket - ARP packet

\* \*/

typedef struct arp\_packet {

ether\_header etherhdr;

ARPHdr arp;

} ARPPacket;

/\*

\* struct ARPTableItem - Cache avaliable neighbors

\* \*/

typedef struct arp\_table\_item {

in\_addr\_t ip;

struct ether\_addr mac;

u\_int16\_t ttl;

} ARPTableItem;

typedef std::map<in\_addr\_t, ARPTableItem> ARPTable;

/\*

\* struct ARPQueueItem - Cache pending datagrams

\* \*/

typedef struct arp\_queue\_item {

packet\_t \*packet;

} ARPQueueItem;

typedef std::map<in\_addr\_t, std::list<ARPQueueItem> > ARPQueue;

class CLink;

class CARP

{

public:

static CARP \* instance()

{

static CARP inst;

return &inst;

}

void init();

~CARP();

/\*

\* Send out network layer datagram

\* \*/

void sendDatagram(packet\_t \*packet);

/\*

\* Send out ARP packet

\* \*/

void sendARP(const struct in\_addr &addr, packet\_t \*packet);

/\*

\* Received ARP packet

\* \*/

void recvARP(packet\_t \*packet);

private:

bool \_isInited;

ARPTable \_arpTable; // arp table cache - Neighbors

ARPQueue \_arpQueue; // arp queue - Pending datagrams wait for ARP resolve

CLink \*\_link;

/\*

\* Cache packet to queue with key.

\*

\* @return The item number has been cached with key.

\* \*/

int cache(const struct in\_addr &key, packet\_t \*packet);

/\*

\* Cache ARP item to table.

\* \*/

void cache(const ARPHdr \*arphdr);

/\*

\* Process cached datagrams when arp reply received

\*

\* @addr The address resolved

\* \*/

void processPendingDatagrams(in\_addr\_t addr, const struct ether\_addr \* mac);

void replyARP(const ARPHdr \*arphdr);

CARP() : \_isInited(false), \_link(nullptr)

{

}

};

//#endif // ARP\_H\_

#pragma once

#include "Neighbor.h"

#include "packet.h"

#include "ip.h"

#include <map>

#include <list>

#include "BaseIO.h"

typedef struct frag\_list {

struct in\_addr saddr, // fragment source ip address

daddr; // fragment destinatin ip address

uint8\_t proto; // upper protocol used

uint16\_t id; // ip header identifier

uint32\_t len; // total length of original datagram

uint32\_t meat; // received length

std::list<packet\_t\*> fragments; // fragment lists(Notice: the pointed block must be deleted correctly)

} IPFragList;

class CNeighbor;

class CNetwork : public CBaseIO

{

public:

//

static CNetwork \* instance()

{

static CNetwork inst;

return &inst;

}

~CNetwork();

void init();

/\*

\* Add ip header, then transport this packet

\*

\* @pkt The packet to send out from local.

\* \*/

int send(packet\_t \*pkt);

/\*

\* Forward a packet. The ip header already in the packet.

\*

\* @pkt The packet to be forwarded

\* \*/

void forward(packet\_t \*pkt);

/\*

\* Deliver received packet to right protocol handler.

\*

\* @pkt The packet to deliver.

\* \*/

void deliver(packet\_t \*pkt);

/\*

\* Received a packet from L2

\*

\* @pkt The received packet

\* \*/

int received(packet\_t \*pkt);

/\*

\* Do defragmentation

\*

\* @iphdr The ip header

\* @pkt The packet to defragment

\* \*/

void defragment(iphdr\_t \*iphdr, packet\_t \*pkt);

/\*

\* Calculate fragment hash code by fragment ip header

\*

\* @iphdr The ip header

\* \*/

uint32\_t fragmentHashCode(iphdr\_t \*iphdr);

/\*

\* Do fragmentation

\*

\* @pkt The packet to fragment

\* @mtu The maximum data length of each fragment

\* \*/

void fragment(packet\_t \*pkt, uint16\_t mtu);

private:

/\*

\* Return an id and increase it

\*

\* @return An Identifier used by ip header

\* \*/

unsigned short getAndIncID(packet\_t \*pkt);

/\*

\* Reassemble a datagram when all fragments available.

\*

\* @pFrags Fragment list pointer

\* \*/

void reasm(IPFragList \*pFrags);

/\*

\* Clear fragment cache.

\*

\* @pFrags Target fragment list pointer

\* \*/

void clear(IPFragList \*pFrags);

CNetwork() : \_neigh(nullptr)

{

}

CNetwork(const CNetwork&);

CNetwork & operator= (const CNetwork&);

CNeighbor \*\_neigh;

CBaseIO \*\_tcp;

CBaseIO \*\_udp;

/\*key: destination ip address, value: id for ip header\*/

std::map<in\_addr\_t, unsigned short> \_idMap;

/\*key: fragment hash code, value: list for datagram fragments\*/

std::map<uint32\_t, IPFragList> \_fragsMap;

};

#pragma once

//#ifndef HARDWARE\_H\_

//#define HARDWARE\_H\_

#include "Device.h"

#include "packet.h"

#include "Link.h"

#include <pcap/pcap.h>

#include <list>

/\* default snap length (maximum bytes per packet to capture) \*/

#define SNAP\_LEN 1518

#define PKT\_BUFF\_TIME 2000

class CLink;

class CHardware

{

public:

//

static CHardware \* instance()

{

static CHardware hardware;

return &hardware;

}

~CHardware()

{

if (\_defaultDev != nullptr && \_defaultDev->handler != nullptr) {

pcap\_close(\_defaultDev->handler);

}

pcap\_freealldevs(\_foundDevs);

}

void init();

void up();

void down();

void transmit(const u\_char\*, size\_t size);

void received();

const Device \* getDefaultDevice() const

{

return \_defaultDev;

}

private:

bool \_isInited;

pcap\_if\_t \*\_foundDevs; // found pcap devices

std::list<Device> \_devs; // avaliable device list

Device \*\_defaultDev; // default device

CLink \*\_link;

int detectDevices(char \*errbuf);

static void getPacket(u\_char \*user, const struct pcap\_pkthdr \*h, const u\_char \*bytes);

CHardware() : \_isInited(false), \_defaultDev(nullptr), \_link(nullptr)

{

}

CHardware(const CHardware&);

CHardware& operator= (const CHardware&);

};

//#endif // HARDWARE\_H\_

#pragma once

#include "packet.h"

/\*

\* BaseIO - Basic input and output interface

\* \*/

class CBaseIO

{

public:

CBaseIO() : \_isInited(false)

{

}

virtual ~CBaseIO()

{

}

virtual void init() = 0;

virtual int send(packet\_t \*pkt) = 0;

virtual int received(packet\_t \*pkt) = 0;

protected:

bool \_isInited;

};

#pragma once

//#ifndef LINK\_H\_

//#define LINK\_H\_

#include "Hardware.h"

#include "Neighbor.h"

#include "Network.h"

#include "packet.h"

#include <memory>

class CNeighbor;

class CHardware;

class CNetwork;

/\*

\* CLink - link layer interface

\*

\* Call neighbor subsystem when send a network datagram.

\* Call hardware when transmit a packet.

\* \*/

class CLink

{

public:

static CLink \* instance()

{

static CLink inst;

return &inst;

}

~CLink()

{

}

void init();

/\*

\* transmit a packet

\* \*/

void transmit(packet\_t \*);

/\*

\* send a network datagram

\* \*/

void send(packet\_t \*packet);

/\*

\* Receive packet from device

\* \*/

void received(const u\_char \*, size\_t);

/\*

\* The default device used to send and receive packet

\* \*/

const Device \* getDefaultDevice() const;

private:

bool \_isInited;

CNetwork \*\_network;

CNeighbor \*\_neigh;

CHardware \*\_hardware;

CLink() : \_isInited(false), \_hardware(nullptr), \_neigh(nullptr)

{

// do not call init() here, will cause '\_\_gnu\_cxx::recursive\_init\_error'

}

CLink(const CLink&);

CLink & operator= (const CLink&);

};

//#endif // LINK\_H\_

#ifndef UDP\_H\_

#define UDP\_H\_

#include <netinet/ip.h>

#define SIZE\_UDP 8

#define SIZE\_PSEUDO\_UDP 12

// udp header

typedef struct sniff\_udp {

uint16\_t uh\_sport; // source port

uint16\_t uh\_dport; // destination port

uint16\_t uh\_len; // udp length

uint16\_t uh\_sum; // udp checksum

} udphdr\_t ;

//pseudo udp header for calculation of check sum

typedef struct pseudo\_udp {

struct in\_addr saddr; // source address

struct in\_addr daddr; // destination address

uint8\_t zero; // zero

uint8\_t protocol; // protocol

uint16\_t len; // UDP length

} pseudo\_udp\_t ;

#endif // UDP\_H\_

#pragma once

//#ifndef IP\_ARP\_H\_

//#define IP\_ARP\_H\_

#include <sys/types.h>

const u\_int8\_t cARPHeaderLen = 28;

/\*

\* ARP header -

\*

\* attribute packed is needed to align one byte.

\* \*/

struct sniff\_arp {

u\_int16\_t htype; // Hardware Type

u\_int16\_t ptype; // Protocol Type

u\_int8\_t hlen; // Hardware Address Length

u\_int8\_t plen; // Protocol Address Length

u\_int16\_t oper; // Operation Code

u\_int8\_t sha[6]; // Sender hardware address

u\_int32\_t spa; // Sender IP address

u\_int8\_t tha[6]; // Target hardware address

u\_int32\_t tpa; // Target IP address

} \_\_attribute\_\_((packed));

typedef struct sniff\_arp ARPHdr;

//#endif // IP\_ARP\_H\_

#pragma once

#include <arpa/inet.h>

extern uint16\_t cksum(const u\_char \*const buf, size\_t size);

#ifndef DEVICE\_H\_

#define DEVICE\_H\_

#include <arpa/inet.h>

#include <netinet/ether.h>

#include <pcap/pcap.h>

#include <iostream>

#include <sstream>

typedef struct device {

u\_int16\_t id;

u\_int16\_t type;

u\_int16\_t mtu; // maximum transmission unit

char \*name;

struct in\_addr ipAddr; // ip

struct ether\_addr hAddr; // mac

pcap\_t \*handler; // pcap packet handler

device ()

{

id = type = 0;

name = nullptr;

handler = nullptr;

}

void show() const

{

std::cout << toString() << std::endl;

}

std::string toString() const

{

std::ostringstream oss;

oss << "Device ["

<< "id=" << id

<< ", type=" << type

<< ", mtu=" << mtu

<< ", name=" << name

<< ", ip=" << inet\_ntoa(ipAddr)

<< ", mac=" << ether\_ntoa(&hAddr)

<< "]";

return oss.str();

}

} Device;

#endif // DEVICE\_H\_

#ifndef UTIL\_H\_

#define UTIL\_H\_

#include <cstdio>

#define DBG\_PREFIX 1

#define DBG\_NEWLINE (1 << 1)

#define DBG\_DEFAULT (DBG\_PREFIX | DBG\_NEWLINE)

#define DBG\_NONE 0

#define DEBUG 1

extern void log(const char \*format, ...);

extern void debug(const char \*format, ...);

extern void debug(int flag, const char \*format, ...);

extern void error(const char \*format, ...);

#define SIZE\_TOK\_BUF 256

typedef struct tok {

int v; // value

const char \* s; // string

} tok\_t ;

extern const char \* tok2str(const tok\_t \* tokp,

const char \* default\_msg,

int v);

class File

{

public:

File(const char \*filename) : file(NULL)

{

if ((file = fopen(filename, "w")) == NULL) {

fprintf(stderr, "Cannot open file: %s, use standard output instead.\n", filename);

file = stdout;

}

else {

fprintf(stdout, "Open file: %s\n", filename);

}

}

~File()

{

if (file != NULL)

fclose(file);

}

FILE \*get() const

{

return file;

}

private:

FILE \*file;

};

#endif // UTIL\_H\_

#include "Neighbor.h"

#include "Util.h"

#define TAG "<Neighbor> "

void CNeighbor::init()

{

if (\_isInited)

return;

\_arp = CARP::instance();

\_arp->init();

\_link = CLink::instance();

\_link->init();

\_isInited = true;

debug(DBG\_DEFAULT, TAG "initied.");

}

void CNeighbor::send(packet\_t \*packet)

{

\_arp->sendDatagram(packet);

}

void CNeighbor::received(packet\_t \*packet)

{

\_arp->recvARP(packet);

}

#include "Network.h"

#include "ip.h"

#include "CheckSum.h"

#include "Util.h"

#include "TCP.h"

#include "UDP.h"

#include <cstring>

#include <algorithm>

#define TAG "<Network> "

const tok\_t ipproto\_values[] = {

{IPPROTO\_TCP, "TCP"},

{IPPROTO\_UDP, "UDP"},

{IPPROTO\_ICMP, "ICMP"},

{IPPROTO\_IP, "IP"},

{0, NULL}

};

/\*

\* A simple implementation of `ip\_select\_fb\_ident()`

\* \*/

uint16\_t CNetwork::getAndIncID(packet\_t \*pkt)

{

in\_addr\_t daddr = pkt->daddr.s\_addr;

uint16\_t id;

if (\_idMap.count(daddr) == 1) {

// find it

id = \_idMap.at(daddr) + 1;

}

else {

// not found, create one

srand((unsigned int)time(NULL));

id = rand() % 0xFFFF;

}

\_idMap[daddr] = id; // update cache

return id;

}

int CNetwork::send(packet\_t \*pkt)

{

log (TAG "%s.\n", \_\_func\_\_);

if (pkt->len > 0xFFFF) {

debug(DBG\_DEFAULT, "Too big packet to send.");

return 0;

}

u\_short ether\_type = 0; // Ethertype value

const char \* etherstr = NULL; // Ethertype string

// Ether dst & src

char dst[MAC\_ASCII\_LEN], src[MAC\_ASCII\_LEN], \*tmp = NULL;

ether\_type = ntohs(ethernet->ether\_type);

printf("Network Layer Protocol: %s (%4X)", tok2str(ethertype\_values, "Unknown", ether\_type),ether\_type );

/\*etherstr = get\_ethertype\_by\_value(ether\_type);\*/

/\*if(etherstr != NULL) {\*/

/\*printf("%s", etherstr);\*/

/\*}\*/

/\*else {\*/

/\*printf("Unknown(%0004X)", ether\_type);\*/

/\*}\*/

/\*

\* The ether\_ntoa() function converts the Ethernet host address addr given in network byte order to a string

\* in standard hex-digits-and-colons notation, omitting leading zeros. The string is returned in a statically

\* allocated buffer, which subsequent calls will overwrite.

\* \*/

tmp = ether\_ntoa((const struct ether\_addr\*)&ethernet->ether\_dhost);

strncpy(dst, tmp, MAC\_ASCII\_LEN);

tmp = ether\_ntoa((const struct ether\_addr\*)&ethernet->ether\_shost);

strncpy(src, tmp, MAC\_ASCII\_LEN);

printf("\n Source: %s\nDestination: %s\n", src, dst);

}

/\*

\* Print ARP header

\* \*/

void print\_arp(const struct sniff\_arp \* arp)

{

int i;

printf("Hardware type: %s\n", ((ntohs(arp->htype) == 1) ? "Ethernet" : "Unknown"));

printf("Protocol type: %s\n", ((ntohs(arp->ptype) == 0x0800) ? "IPv4" : "Unknown"));

printf("Operation: %s\n", ((ntohs(arp->oper) == ARPOP\_REQUEST) ? "ARP Request" : "ARP Reply"));

//If is Ethernet and IPv4, print packet contents

if( ntohs(arp->htype) == 1 && ntohs(arp->ptype) == 0x0800) {

printf("Sender MAC: ");

for(i=0; i<6 ; ++i)

printf("%02X:", arp->sha[i]);

printf("\nSender IP: ");

struct in\_addr spa;

spa.s\_addr = arp->spa;

printf("%s", inet\_ntoa(spa));

/\*for(i=0; i<4 ; ++i)\*/

/\*printf("%d.", arp->spa[i]);\*/

printf("\nTarget MAC: ");

for(i=0; i<6 ; ++i)

printf("%02X:", arp->tha[i]);

printf("\nTarget IP: ");

struct in\_addr tpa;

tpa.s\_addr = arp->tpa;

printf("%s", inet\_ntoa(tpa));

/\*for(i=0; i<4 ; ++i)\*/

/\*printf("%d.", arp->tpa[i]);\*/

printf("\n");

}

}

void print\_tcp(const struct sniff\_tcp \* tcp,

const struct sniff\_ip \* ip,

const u\_char \* packet)

{

const char \*payload; /\* Packet payload \*/

int size\_ip = IP\_HL(ip)\*4;

int size\_tcp = TH\_OFF(tcp)\*4;

int size\_payload;

printf(" Src port: %d\n", ntohs(tcp->th\_sport));

printf(" Dst port: %d\n", ntohs(tcp->th\_dport));

/\* define/compute tcp payload (segment) offset \*/

//payload = (u\_char \*)(packet + SIZE\_ETHERNET + size\_ip + size\_tcp);

payload = (const char \*)(packet + SIZE\_ETHERNET + size\_ip + size\_tcp);

/\* compute tcp payload (segment) size \*/

size\_payload = ntohs(ip->ip\_len) - (size\_ip + size\_tcp);

/\*

\* Print payload data; it might be binary, so don't just

\* treat it as a string.

\*/

printf(" Payload (%d bytes):\n", size\_payload);

if (size\_payload > 0) {

/\*printf(" Payload (%d bytes):\n", size\_payload);\*/

//print\_payload(payload, size\_payload);

print\_payload((const unsigned char\*)payload, size\_payload);

}

}

void print\_udp(const struct sniff\_udp \* udp,

const struct sniff\_ip \* ip,

const u\_char \* packet)

{

const char \* payload;

int size\_ip = IP\_HL(ip)\*4;

int size\_udp = 8;

int size\_payload;

printf(" Src port: %d\n", ntohs(udp->uh\_sport));

printf(" Dst port: %d\n", ntohs(udp->uh\_dport));

//payload = (u\_char \*)(packet + SIZE\_ETHERNET + size\_ip + size\_udp);

payload = (const char \*)(packet + SIZE\_ETHERNET + size\_ip + size\_udp);

size\_payload = ntohs(udp->uh\_len) - size\_udp;

printf(" Payload (%d bytes):\n", size\_payload);

if(size\_payload > 0) {

//print\_payload(payload, size\_payload);

print\_payload((const u\_char\*)payload, size\_payload);

}

}

void print\_icmp(const struct sniff\_icmp \* icmp,

const struct sniff\_ip \* ip,

const u\_char \* packet)

{

uint8\_t type;

uint8\_t code;

type = icmp->icmp\_type;

printf("Message Type: %s (%d)\n", tok2str(icmptype\_values, "Unknown", type), type);

code = icmp->icmp\_code;

printf("Message Code: %s (%d)\n", tok2str(icmpcode\_values, "Unknown", code), code);

}

/\*

\* Handle ARP header

\* \*/

void handle\_arp(const struct sniff\_arp \* arp)

{

print\_arp(arp);

}

/\*

\* Handle IP datagram

\* \*/

void handle\_ip(const struct sniff\_ip \* ip,

const struct pcap\_pkthdr \* header,

const u\_char \* packet)

{

const struct sniff\_tcp \*tcp; /\* The TCP header \*/

const struct sniff\_udp \*udp; /\* The UDP header \*/

const struct sniff\_icmp \*icmp; /\* The ICMP header \*/

const char \*payload; /\* Packet payload \*/

int size\_ip = IP\_HL(ip)\*4;

int size\_tcp;

int size\_udp;

int size\_payload;

/\* print source and destination IP addresses \*/

printf(" From: %s\n", inet\_ntoa(ip->ip\_src));

printf(" To: %s\n", inet\_ntoa(ip->ip\_dst));

/\* determine protocol \*/

switch(ip->ip\_p) {

case IPPROTO\_TCP:

printf(" Protocol: TCP\n");

/\* define/compute tcp header offset \*/

tcp = (struct sniff\_tcp\*)(packet + SIZE\_ETHERNET + size\_ip);

size\_tcp = TH\_OFF(tcp)\*4;

if (size\_tcp < 20) {

printf(" \* Invalid TCP header length: %u bytes\n", size\_tcp);

}

else {

print\_tcp(tcp, ip, packet);

}

break;

case IPPROTO\_UDP:

printf(" Protocol: UDP\n");

udp = (const udphdr\_t\*)(packet + SIZE\_ETHERNET + size\_ip);

print\_udp(udp, ip, packet);

return;

case IPPROTO\_ICMP:

printf(" Protocol: ICMP\n");

icmp = (const icmphdr\_t \*)(packet + SIZE\_ETHERNET + size\_ip);

print\_icmp(icmp, ip, packet);

break;

case IPPROTO\_IP:

printf(" Protocol: IP\n");

return;

default:

printf(" Protocol: unknown\n");

return;

}

}

/\*

\* Handle ethernet packet

\* \*/

void handle\_ether(const struct sniff\_ethernet \* ethernet,

const struct pcap\_pkthdr \* header,

const u\_char \* packet)

{

const struct sniff\_arp \*arp; /\* The ARP header \*/

const struct sniff\_ip \*ip; /\* The IP header \*/

int size\_ip;

u\_short ether\_type;

/\* Print ethernet informations \*/

print\_ether(ethernet);

/\* Handle packet according to ether\_type \*/

ether\_type = ntohs(ethernet->ether\_type);

switch(ether\_type) {

case ETHERTYPE\_ARP: {

arp = (const struct sniff\_arp\*)(packet + SIZE\_ETHERNET);

handle\_arp(arp);

break;

}

case ETHERTYPE\_REVARP: { break; }

case ETHERTYPE\_IP: {

/\* define/compute ip header offset \*/

ip = (const struct sniff\_ip\*)(packet + SIZE\_ETHERNET);

size\_ip = IP\_HL(ip)\*4;

if (size\_ip < 20) {

printf(" \* Invalid IP header length: %u bytes\n", size\_ip);

}

else {

handle\_ip(ip, header, packet);

}

break;

}

case ETHERTYPE\_IPV6: { break; }

case ETHERTYPE\_LOOPBACK: { break; }

default:

break;

}

/\*ip = (struct sniff\_ip\*)(packet + SIZE\_ETHERNET);\*/

/\*size\_ip = IP\_HL(ip)\*4;\*/

/\*if (size\_ip < 20) {\*/

/\*printf(" \* Invalid IP header length: %u bytes\n", size\_ip);\*/

/\*return;\*/

/\*}\*/

}

/\*

\* dissect/print packet

\*/

void

got\_packet(u\_char \*args, const struct pcap\_pkthdr \*header, const u\_char \*packet)

{

static int count = 1; /\* packet counter \*/

/\* declare pointers to packet headers \*/

const struct sniff\_ethernet \*ethernet; /\* The ethernet header [1] \*/

printf("\nPacket number %d:\n", count);

printf("Packet len: %d\n", header->len);

count++;

/\* define ethernet header \*/

ethernet = (struct sniff\_ethernet\*)(packet);

handle\_ether(ethernet, header, packet);

return;

}

int main(int argc, char \*\*argv)

{

char \*dev = NULL; /\* capture device name \*/

char errbuf[PCAP\_ERRBUF\_SIZE]; /\* error buffer \*/

pcap\_t \*handle; /\* packet capture handle \*/

char filter\_exp[] = "arp"; /\* filter expression [3] \*/

/\*char filter\_exp[] = ""; [> filter expression [3] <]\*/

struct bpf\_program fp; /\* compiled filter program (expression) \*/

bpf\_u\_int32 mask; /\* subnet mask \*/

bpf\_u\_int32 net; /\* ip \*/

int num\_packets = 10; /\* number of packets to capture \*/

print\_app\_banner();

/\* check for capture device name on command-line \*/

if (argc == 2) {

dev = argv[1];

}

else if (argc > 2) {

fprintf(stderr, "error: unrecognized command-line options\n\n");

print\_app\_usage();

exit(EXIT\_FAILURE);

}

else {

/\* find a capture device if not specified on command-line \*/

dev = pcap\_lookupdev(errbuf);

if (dev == NULL) {

fprintf(stderr, "Couldn't find default device: %s\n",

errbuf);

exit(EXIT\_FAILURE);

}

}

/\* get network number and mask associated with capture device \*/

if (pcap\_lookupnet(dev, &net, &mask, errbuf) == -1) {

fprintf(stderr, "Couldn't get netmask for device %s: %s\n",

dev, errbuf);

net = 0;

mask = 0;

}

/\* print capture info \*/

printf("Device: %s\n", dev);

printf("Number of packets: %d\n", num\_packets);

printf("Filter expression: %s\n", filter\_exp);

/\* open capture device \*/

handle = pcap\_open\_live(dev, SNAP\_LEN, 1, 1000, errbuf);

if (handle == NULL) {

fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);

exit(EXIT\_FAILURE);

}

/\* make sure we're capturing on an Ethernet device [2] \*/

if (pcap\_datalink(handle) != DLT\_EN10MB) {

fprintf(stderr, "%s is not an Ethernet\n", dev);

exit(EXIT\_FAILURE);

}

/\* compile the filter expression \*/

if (pcap\_compile(handle, &fp, filter\_exp, 0, net) == -1) {

fprintf(stderr, "Couldn't parse filter %s: %s\n",

filter\_exp, pcap\_geterr(handle));

exit(EXIT\_FAILURE);

}

/\* apply the compiled filter \*/

if (pcap\_setfilter(handle, &fp) == -1) {

fprintf(stderr, "Couldn't install filter %s: %s\n",

filter\_exp, pcap\_geterr(handle));

exit(EXIT\_FAILURE);

}

/\* now we can set our callback function \*/

pcap\_loop(handle, num\_packets, got\_packet, NULL);

/\* cleanup \*/

pcap\_freecode(&fp);

pcap\_close(handle);

printf("\nCapture complete.\n");

return 0;

}

/\*

\* DemoTimeServer - UDP Demo

\*

\* Receive client message and reply current system time.

\* \*/

#include "Socket.h"

#include <string>

#include <cstring>

#include <time.h>

const int cPort = 1618;

const int cBufSize = 4096;

int main(int argc, char\* argv[])

{

if (argc < 2) {

printf("Usage: %s <ip> <port>.\n", argv[0]);

return (0);

}

int sockfd;

struct sockaddr\_in svrAddr;

CSocket socket;

if ((sockfd = socket.socket(AF\_INET, SOCK\_DGRAM, 0)) == -1) {

printf("Failed socket().\n");

return 0;

}

svrAddr.sin\_family = AF\_INET;

inet\_aton(argv[1], &svrAddr.sin\_addr);

if (argc > 2) {

svrAddr.sin\_port = htons(std::stoi(argv[2]));

}

else {

svrAddr.sin\_port = htons(cPort);

}

if (socket.bind((struct sockaddr\*)&svrAddr, sizeof(struct sockaddr)) == -1) {

fprintf(stderr, "Failed bind().\n");

return 0;

}

else {

printf("%s run with %s:%d.\n\n", argv[0], argv[1], ntohs(svrAddr.sin\_port));

}

struct sockaddr\_in client;

unsigned int addrlen = sizeof(client);

char buf[cBufSize];

while (true) {

int bytesRecv = socket.recvfrom(buf, cBufSize, 0, (struct sockaddr\*)&client, &addrlen);

if (bytesRecv > 0) {

// print received message

buf[bytesRecv] = '\0';

printf("Server received: %s.\n", buf);

// reply client

time\_t now = time(NULL);

snprintf(buf, cBufSize, "%.24s\r\n", ctime(&now));

socket.sendto(buf, strlen(buf), 0, (struct sockaddr\*)&client, sizeof(struct sockaddr));

printf("Replied current time.\n");

}

else {

printf("Server receive error, bytesRecv = %d.\n", bytesRecv);

}

}

return 0;

}

#include "Socket.h"

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <unistd.h>

#include <errno.h>

#include <signal.h>

#include <sys/ipc.h>

#include <sys/stat.h>

#include <sys/shm.h>

#include "Util.h"

#define TAG "<CSocket> "

int sig; // received signal

void handler1(int signo)

{

printf("Received signal SIGUSR1(%d).\n", signo);

sig = signo;

}

void handler2(int signo)

{

sig = signo;

printf("Received signal SIGUSR2(%d).\n", signo);

}

int CSocket::waitForSuccess(int signo)

{

pause();

int success;

if (sig == signo) {

success = \*((int \*)\_pBlock->buf1);

}

else {

success = -1;

}

return success;

}

CSocket::CSocket()

{

log(TAG "New socket created.\n");

attachSharedMem();

}

CSocket::~CSocket()

{

close();

detachSharedMem();

log(TAG "A socket destroied.\n");

}

void CSocket::attachSharedMem()

{

key\_t key; // shared memory key

struct shmid\_ds buf;

signal(SIGUSR1, handler1); // SIGUSR1 for commands

signal(SIGUSR2, handler2); // SIGUSR2 for datas

// prepare a key

if ((key = ftok(cKeyPath, cKeyID)) == -1) {

printf("Failed ftok().\n");

}

// try attach Protocol process created shared memory

if ((\_shmid = shmget(key, cSHMSize, IPC\_CREAT | IPC\_EXCL)) == -1) {

if (errno == EEXIST) {

if ((\_shmid = shmget(key, cSHMSize, IPC\_CREAT | S\_IRUSR | S\_IWUSR)) == -1) {

printf("Failed shmget(): %s.\n", strerror(errno));

exit(EXIT\_FAILURE);

}

else {

printf("Success attach shared memory %d.\n", \_shmid);

}

}

else {

// no shared memory exist, maybe Protocol not started, exit...

fprintf(stderr, "Unable to attach shared memory.\n");

exit(0);

}

}

else {

if (shmctl(\_shmid, IPC\_RMID, 0) == -1) {

printf("Failed shmctl(): %s.\n", strerror(errno));

}

printf("Protocol not started, quit...\n");

exit(0);

}

// get shared memory address

if ((\_pBlock = (SharedBlock \*)shmat(\_shmid, 0, 0)) == (void \*)-1) {

printf("Failed shmat(): %s.\n", strerror(errno));

}

printf("pBlock: %p, pBlock->buf1: %p, pBlock->buf2: %p\n", \_pBlock, \_pBlock->buf1, \_pBlock->buf2);

// get created process id

if (shmctl(\_shmid, IPC\_STAT, &buf) == -1) {

printf("Failed shmctl().\n");

}

else {

printf("Protocol process: %d.\n", buf.shm\_cpid);

\_protoPid = buf.shm\_cpid;

}

// do work here

//printf("Shared: %s\n", \_pBlock->buf1);

//kill(buf.shm\_cpid, SIGUSR1);

}

void CSocket::detachSharedMem()

{

if (shmdt(\_pBlock) == -1) {

printf("Failed shmdt(): %s.\n", strerror(errno));

}

}

int CSocket::init(int family, int type, int protocol)

{

\_sock.pid = getpid();

\_sock.sockfd = \_sock.pid;

\_sock.family = family;

\_sock.type = type;

\_sock.protocol = protocol;

\_sock.addr.s\_addr = 0;

\_sock.port = 0;

SockPacket sockPkt;

sockPkt.type = SOCK\_CREATE;

memcpy(sockPkt.data, &\_sock, sizeof(Sock));

// Copy to shared memory and notify this

memcpy(\_pBlock->buf2, &sockPkt, sizeof(Sock) + sizeof(SockPktType));

kill(\_protoPid, SIGUSR1);

log(TAG "%s : kill signal SIGUSR1 to process %d.\n", \_\_func\_\_, \_protoPid);

int result = waitForSuccess(SIGUSR1);

printf("Created socket: %d\n", result);

return result;

}

int CSocket::socket(int family, int type, int protocol)

{

return init(family, type, protocol);

}

int CSocket::bind(const struct sockaddr\* addr, socklen\_t len)

{

struct sockaddr\_in bindAddr = \*((struct sockaddr\_in \*)addr);

\_sock.addr = bindAddr.sin\_addr;

\_sock.port = bindAddr.sin\_port;

SockPacket sockPkt;

sockPkt.type = SOCK\_BIND;

memcpy(sockPkt.data, &\_sock, sizeof(\_sock));

memcpy(\_pBlock->buf2, &sockPkt, sizeof(SockPktType) + sizeof(Sock));

kill(\_protoPid, SIGUSR1);

pause();

if (sig == SIGUSR1) {

int success = \*((int \*)\_pBlock->buf1);

if (success == 1) {

return 0;

}

else {

\_sock.addr.s\_addr = 0;

\_sock.port = 0;

return -1;

}

}

else {

\_sock.addr.s\_addr = 0;

\_sock.port = 0;

fprintf(stderr, "Not SIGUSR1 received\n");

return -1;

}

}

int CSocket::sendto(const char\* buf, size\_t len, int flags,

const struct sockaddr\* dstAddr, socklen\_t addrlen)

{

// todo: Send to ProtoSocket send message

// data format: ProtoSocket{type, {SockData, buf}}

// or: ProtoSocket{type, {left buf}}

SockDataHdr sockDataHdr;

sockDataHdr.sockfd = \_sock.sockfd;

sockDataHdr.dstAddr = \*((struct sockaddr\_in \*)dstAddr);

sockDataHdr.flag = flags;

sockDataHdr.len = len;

SockPacket sockPkt;

sockPkt.type = SOCK\_SENDTO;

char \*pData = sockPkt.data;

memcpy(pData, &sockDataHdr, sizeof(SockDataHdr));

pData += sizeof(SockDataHdr);

int bufLeft = cSHMDataSize - sizeof(SockDataHdr);

int dataLeft = len;

while (dataLeft > 0) {

int dataLen;

if (bufLeft <= 0) { // re-point to buffer start

bufLeft = cSHMDataSize;

pData = sockPkt.data;

}

if (dataLeft <= bufLeft) {

dataLen = dataLeft;

}

else {

dataLen = bufLeft;

}

memcpy(pData, buf, dataLen);

pData += dataLen;

dataLeft -= dataLen;

bufLeft -= dataLen;

int bytes = pData - sockPkt.data + sizeof(SockPktType);

printf("will copy %d bytes.\n", bytes);

memcpy(\_pBlock->buf2, &sockPkt, bytes);

kill(\_protoPid, SIGUSR2);

}

int byteSend = waitForSuccess(SIGUSR2);

printf("Send %d bytes.\n", byteSend);

return byteSend;

}

int CSocket::recvfrom(char\* buf, size\_t len, int flags,

struct sockaddr\* srcAddr, socklen\_t\* addrlen)

{

SockDataHdr dataHdr;

dataHdr.sockfd = \_sock.sockfd;

dataHdr.flag = flags;

dataHdr.len = len;

SockPacket sockPkt;

sockPkt.type = SOCK\_RECVFROM;

memcpy(sockPkt.data, &dataHdr, sizeof(dataHdr));

memcpy(\_pBlock->buf2, &sockPkt, sizeof(SockPktType) + sizeof(dataHdr));

kill(\_protoPid, SIGUSR1);

pause();

// read data from ProtoSocket and set value-result parameters

char \*pData = \_pBlock->buf1;

SockDataHdr\* rcvDataHdr = (SockDataHdr \*)pData;

struct sockaddr\_in\* fromAddr = &rcvDataHdr->srcAddr;

\*srcAddr = \*((struct sockaddr\*)&rcvDataHdr->srcAddr);

\*addrlen = sizeof(struct sockaddr);

printf("Received data from %s:%d.\n", inet\_ntoa(fromAddr->sin\_addr), ntohs(fromAddr->sin\_port));

pData += sizeof(SockDataHdr);

int dataLen = len;

if (dataLen > rcvDataHdr->len) {

dataLen = rcvDataHdr->len;

}

memcpy(buf, pData, dataLen);

return dataLen;

}

int CSocket::close()

{

// there is no need to close an unconnected socket

if (\_sock.type == SOCK\_STREAM && \_sock.state == SS\_UNCONNECTED) {

log(TAG "%s(): stream socket not connected, there is no need to close\n", \_\_func\_\_);

return 0;

}

SockPacket sockPkt;

sockPkt.type = SOCK\_CLOSE;

memcpy(sockPkt.data, &\_sock, sizeof(Sock));

memcpy(\_pBlock->buf2, &sockPkt, sizeof(SockPktType) + sizeof(Sock));

kill(\_protoPid, SIGUSR1);

return waitForSuccess(SIGUSR1) - 1;

}

int CSocket::connect(const struct sockaddr\* addr, socklen\_t len)

{

SockPacket sockPkt;

sockPkt.type = SOCK\_CONNECT;

struct sockaddr\_in \* dstAddr = (struct sockaddr\_in \*)addr;

\_sock.peerAddr = dstAddr->sin\_addr;

\_sock.peerPort = dstAddr->sin\_port;

memcpy(sockPkt.data, &\_sock, sizeof(\_sock));

memcpy(\_pBlock->buf2, &sockPkt, sizeof(SockPktType) + sizeof(\_sock));

kill(\_protoPid, SIGUSR1);

pause();

Sock \*sock = (Sock \*)\_pBlock->buf1;

if (sock->port > 0) {

\_sock.addr = sock->addr;

\_sock.port = sock->port;

\_sock.state = sock->state;

return 0;

}

else {

return -1;

}

}

SockDataHdr CSocket::makeDataHeader(size\_t len, int flag)

{

SockDataHdr sockDataHdr;

sockDataHdr.sockfd = \_sock.sockfd;

sockDataHdr.flag = flag;

sockDataHdr.len = len;

struct sockaddr\_in srcAddr, dstAddr;

srcAddr.sin\_addr = \_sock.addr;

srcAddr.sin\_port = \_sock.port;

dstAddr.sin\_addr = \_sock.peerAddr;

dstAddr.sin\_port = \_sock.peerPort;

sockDataHdr.srcAddr = srcAddr;

sockDataHdr.dstAddr = dstAddr;

return sockDataHdr;

}

int CSocket::send(const char \* buf, size\_t len, int flag)

{

log(TAG "%s(): \"%s\"\n", \_\_func\_\_, buf);

log(TAG "%s(): addr %s:%d\n", \_\_func\_\_, inet\_ntoa(\_sock.addr), ntohs(\_sock.port));

log(TAG "%s(): peer %s:%d\n", \_\_func\_\_, inet\_ntoa(\_sock.peerAddr), ntohs(\_sock.peerPort));

// todo: Send to ProtoSocket send message

// data format: ProtoSocket{type, {SockData, buf}}

// or: ProtoSocket{type, {left buf}}

SockDataHdr sockDataHdr = makeDataHeader(len, flag);

SockPacket sockPkt;

sockPkt.type = SOCK\_SEND;

char \*pData = sockPkt.data;

memcpy(pData, &sockDataHdr, sizeof(SockDataHdr));

pData += sizeof(SockDataHdr);

int bufLeft = cSHMDataSize - sizeof(SockDataHdr);

int dataLeft = len;

while (dataLeft > 0) {

int dataLen;

if (bufLeft <= 0) { // re-point to buffer start

bufLeft = cSHMDataSize;

pData = sockPkt.data;

}

if (dataLeft <= bufLeft) {

dataLen = dataLeft;

}

else {

dataLen = bufLeft;

}

memcpy(pData, buf, dataLen);

pData += dataLen;

dataLeft -= dataLen;

bufLeft -= dataLen;

int bytes = pData - sockPkt.data + sizeof(SockPktType);

printf("will copy %d bytes.\n", bytes);

memcpy(\_pBlock->buf2, &sockPkt, bytes);

kill(\_protoPid, SIGUSR2);

}

int byteSend = waitForSuccess(SIGUSR2);

printf("Send %d bytes.\n", byteSend);

return byteSend;

}

int CSocket::recv(char \* buf, size\_t len, int flag)

{

log(TAG "%s(): addr %s:%d\n", \_\_func\_\_, inet\_ntoa(\_sock.addr), ntohs(\_sock.port));

log(TAG "%s(): peer %s:%d\n", \_\_func\_\_, inet\_ntoa(\_sock.peerAddr), ntohs(\_sock.peerPort));

// each recv() should return as soon as possible, if there is no data, -1 is returned

SockDataHdr sockDataHdr = makeDataHeader(len, flag);

SockPacket sockPkt;

sockPkt.type = SOCK\_RECV;

memcpy(sockPkt.data, &sockDataHdr, sizeof(SockDataHdr));

memcpy(\_pBlock->buf2, &sockPkt, sizeof(SockPktType) + sizeof(SockDataHdr));

uint8\_t leftChance = 3;

kill(\_protoPid, SIGUSR2);

pause();

// read data from ProtoSocket and set value-result parameters

char \*pData = \_pBlock->buf1;

SockDataHdr\* rcvDataHdr = (SockDataHdr \*)pData;

//while (leftChance-- > 0) {

while (true) {

if (rcvDataHdr->len <= 0) {

// try again if we still get chance

//log(TAG "%s(): no data available, left try times=%d.\n", \_\_func\_\_, leftChance);

log(TAG "%s(): try again\n", \_\_func\_\_);

sleep(1);

kill(\_protoPid, SIGUSR2);

pause();

}

else {

break;

}

}

if (leftChance <= 0 && rcvDataHdr->len <= 0) {

log(TAG "%s(): return without data\n", \_\_func\_\_);

return -1;

}

pData += sizeof(SockDataHdr);

int dataLen = len;

if (dataLen > rcvDataHdr->len) {

dataLen = rcvDataHdr->len;

}

memcpy(buf, pData, dataLen);

return dataLen;

}

int CSocket::listen(int backlog)

{

SockPacket sockPkt;

sockPkt.type = SOCK\_LISTEN;

char \*pData = sockPkt.data;

memcpy(pData, &\_sock, sizeof(\_sock));

pData += sizeof(\_sock);

memcpy(pData, &backlog, sizeof(int));

memcpy(\_pBlock->buf2, &sockPkt, sizeof(SockPktType) + sizeof(\_sock) + sizeof(int));

kill(\_protoPid, SIGUSR1);

return waitForSuccess(SIGUSR1) - 1;

}

std::unique\_ptr<CSocket> CSocket::accept(struct sockaddr \* sockaddr, socklen\_t \* addrlen)

{

SockPacket sockPkt;

sockPkt.type = SOCK\_ACCEPT;

memcpy(sockPkt.data, &\_sock, sizeof(\_sock));

memcpy(\_pBlock->buf2, &sockPkt, sizeof(SockPktType) + sizeof(Sock));

kill(\_protoPid, SIGUSR1);

pause();

Sock \*sock = (Sock \*)\_pBlock->buf1;

std::unique\_ptr<CSocket> pSock(new CSocket());

pSock->\_sock = \*sock;

struct sockaddr\_in \*fromAddr = (struct sockaddr\_in \*)sockaddr;

fromAddr->sin\_addr = sock->peerAddr;

fromAddr->sin\_port = sock->peerPort;

fromAddr->sin\_family = sock->family;

memset(fromAddr->sin\_zero, 0, sizeof(fromAddr->sin\_zero));

\*addrlen = sizeof(struct sockaddr\_in);

return pSock;

}

#include "ProtoSocket.h"

#include <unistd.h>

#include <sys/stat.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <signal.h>

#include <algorithm>

#include "UDP.h"

#include "TCP.h"

#include "Network.h"

#include "Link.h"

#include "Hardware.h"

#include "Util.h"

#define TAG "<CProtoSocket> "

using std::string;

int sig; // signal received

void handler0(int signo)

{

sig = signo;

printf("Received signal INT.\n");

}

void handler1(int signo)

{

sig = signo;

printf("Received signal USR1.\n");

}

void handler2(int signo)

{

sig = signo;

printf("Received signal USR2.\n");

}

CProtoSocket::CProtoSocket()

{

log(TAG "%s(): sizeof(SockPktType)=%d.\n", \_\_func\_\_, sizeof(SockPktType));

createSharedMem();

}

CProtoSocket::~CProtoSocket()

{

destroySharedMem();

printf("CProtoSocket destructed.\n");

}

void CProtoSocket::createSharedMem()

{

key\_t key;

if ((key = ftok(cKeyPath, cKeyID)) == -1) {

fprintf(stderr, "Failed ftok().\n");

}

if ((\_shmid = shmget(key, cSHMSize, IPC\_CREAT | IPC\_EXCL | S\_IRUSR | S\_IWUSR |

S\_IRGRP | S\_IWGRP |

S\_IROTH | S\_IWOTH)) == -1) {

fprintf(stderr, "Failed shmget().\n");

exit(EXIT\_FAILURE);

}

else {

printf("Created shared memory %d.\n", \_shmid);

}

if ((\_pBlock = (SharedBlock \*)shmat(\_shmid, 0, 0)) == (void \*)-1) {

fprintf(stderr, "Failed shmat().\n");

}

else {

printf("pBlock: %p, pBlock->buf1: %p, pBlock->buf2: %p\n",

\_pBlock, \_pBlock->buf1, \_pBlock->buf2);

}

}

void CProtoSocket::destroySharedMem()

{

if (shmdt(\_pBlock) == -1) {

fprintf(stderr, "Failed shmdt().\n");

}

if (shmctl(\_shmid, IPC\_RMID, 0) == -1) {

fprintf(stderr, "Failed shmctl().\n");

}

}

void CProtoSocket::run()

{

// protocols init, must in order : top-to-down

CTCP::instance()->init();

CUDP::instance()->init();

CNetwork::instance()->init();

CLink::instance()->init();

CHardware::instance()->init();

// catch signals

signal(SIGUSR1, handler1);

signal(SIGUSR2, handler2);

signal(SIGINT, handler0);

printf("Protocol socket running...\n");

while (true) {

pause();

if (sig == SIGUSR1 || sig == SIGUSR2) {

handleSockRequest();

// SIGUSR1 do command work,

// SIGUSR2 do data work

}

else if (sig == SIGINT){

break;

}

else {

printf("Unknown signal: %d.\n", sig);

}

sig = 0;

printf("Protocols alive...\n");

}

printf("Protocol socket exit...\n");

}

void CProtoSocket::handleSockRequest()

{

SockPacket \*sockPkt;

sockPkt = (SockPacket \*)\_pBlock->buf2;

switch (sockPkt->type) {

case SOCK\_CREATE:

{

handleCreate(sockPkt); break;

}

case SOCK\_BIND:

{

handleBind(sockPkt); break;

}

case SOCK\_SENDTO:

{

handleSendTo(sockPkt); break;

}

case SOCK\_SEND:

{

handleSend(sockPkt); break;

}

case SOCK\_RECVFROM:

{

handleRecvFrom(sockPkt); break;

}

case SOCK\_RECV:

{

handleRecv(sockPkt); break;

}

case SOCK\_CONNECT:

{

handleConnect(sockPkt); break;

}

case SOCK\_LISTEN:

{

handleListen(sockPkt); break;

}

case SOCK\_ACCEPT:

{

handleAccept(sockPkt); break;

}

case SOCK\_CLOSE:

{

handleClose(sockPkt); break;

}

default:

fprintf(stderr, "Unkonwn socket packet type: %d.\n", sockPkt->type);

break;

}

}

void CProtoSocket::handleCreate(SockPacket \*sockPkt)

{

Sock \*sock;

sock = (Sock \*)sockPkt->data;

printf("pid: %d, family: %d, type: %d, protocol: %d\n",

sock->pid, sock->family, sock->type, sock->protocol);

sock->state = SS\_UNCONNECTED;

InetSock sk{

.\_sock = \*sock,

.sk\_state = CLOSED

};

// save to socket pool

\_sockPool.emplace(sock->sockfd, sk);

// write back sockfd

memcpy(\_pBlock->buf1, &sock->sockfd, sizeof(int));

afterHandle(sock->pid, SIGUSR1, \_\_func\_\_);

}

void CProtoSocket::handleBind(SockPacket \*sockPkt)

{

Sock \*sock = (Sock \*)sockPkt->data;

InetSock &cached = \_sockPool.at(sock->sockfd);

cached.sk\_addr = sock->addr;

cached.sk\_port = sock->port;

int success = 1;

memcpy(\_pBlock->buf1, &success, sizeof(success));

afterHandle(cached.sk\_pid, SIGUSR1, \_\_func\_\_);

//kill(cached.pid, SIGUSR1);

//log(TAG "%s : kill signal SIGUSR1 to process %d.\n", \_\_func\_\_, cached.pid);

}

void CProtoSocket::handleSendTo(SockPacket \*sockPkt)

{

SockDataHdr \*sockDataHdr;

sockDataHdr = (SockDataHdr \*)sockPkt->data;

struct sockaddr\_in \*dstAddr = (struct sockaddr\_in \*)&sockDataHdr->dstAddr;

printf("socket: %d want to send %d bytes data to %s:%d.\n",

sockDataHdr->sockfd, sockDataHdr->len,

inet\_ntoa(dstAddr->sin\_addr), ntohs(dstAddr->sin\_port));

// get data to send

char \*pData = sockPkt->data;

pData += sizeof(SockDataHdr);

// ---- debug only

char \*buf = (char \*)malloc(sockDataHdr->len + 1);

memcpy(buf, pData, sockDataHdr->len);

buf[sockDataHdr->len] = '\0';

printf("Contents to send: %s.\n", buf);

free(buf);

// ---- /debug only

packet\_t pkt;

pkt.buf = (unsigned char\*)pData;

pkt.size = sockDataHdr->len;

pkt.daddr = dstAddr->sin\_addr;

pkt.dport = dstAddr->sin\_port;

// get this socket

InetSock & sock = \_sockPool.at(sockDataHdr->sockfd);

// set local address as needed

setLocalAddr(&sock);

pkt.saddr = sock.sk\_addr;

pkt.sport = sock.sk\_port;

// call UDP::send()

CUDP::instance()->send(&pkt);

//

// notice, this code assume data will not overflow the buffer size

// to handle the overflow situation, modify this code

// notify send bytes

memcpy(\_pBlock->buf1, &sockDataHdr->len, sizeof(int));

afterHandle(sock.sk\_pid, SIGUSR2, \_\_func\_\_);

}

void CProtoSocket::handleRecvFrom(SockPacket \*sockPkt)

{

SockDataHdr \*dataHdr = (SockDataHdr \*)sockPkt->data;

log(TAG "socket %d wanna recvfrom max %d bytes data.\n", dataHdr->sockfd, dataHdr->len);

// get this socket

InetSock& sock = \_sockPool.at(dataHdr->sockfd);

// add pending recvfrom socket

//\_pendingSocks.emplace(dataHdr->sockfd, sock.port);

\_pendingSocks.emplace(&sock);

log(TAG "%s : add penging socket %d:%d.\n", \_\_func\_\_, sock.sk\_sockfd, ntohs(sock.sk\_port));

}

void CProtoSocket::handleSend(SockPacket \*sockPkt)

{

SockDataHdr \*sockDataHdr;

sockDataHdr = (SockDataHdr \*)sockPkt->data;

struct sockaddr\_in \*srcAddr = (struct sockaddr\_in \*)&sockDataHdr->srcAddr;

struct sockaddr\_in \*dstAddr = (struct sockaddr\_in \*)&sockDataHdr->dstAddr;

string key = CTCP::instance()->keyOf(srcAddr->sin\_addr, srcAddr->sin\_port, dstAddr->sin\_addr, dstAddr->sin\_port);

// find connection first

log(TAG "%s(): %s\n", \_\_func\_\_, key.c\_str());

ConnPMap::iterator it = \_connPPool.find(key);

if (it != \_connPPool.end()) {

// get data to send

char \*pData = sockPkt->data;

pData += sizeof(SockDataHdr);

// ---- debug only

char \*buf = (char \*)malloc(sockDataHdr->len + 1);

memcpy(buf, pData, sockDataHdr->len);

buf[sockDataHdr->len] = '\0';

log(TAG "%s() contents send: %s.\n", \_\_func\_\_, buf);

free(buf);

// ---- /debug only

packet\_t pkt;

pkt.buf = (unsigned char\*)pData;

pkt.size = sockDataHdr->len;

pkt.saddr = srcAddr->sin\_addr;

pkt.sport = srcAddr->sin\_port;

pkt.daddr = dstAddr->sin\_addr;

pkt.dport = dstAddr->sin\_port;

CTCP::instance()->send(&pkt);

// notice, this code assume data will not overflow the buffer size

// to handle the overflow situation, fix this code

memcpy(\_pBlock->buf1, &sockDataHdr->len, sizeof(int));

afterHandle(it->second->ics\_pid, SIGUSR2, \_\_func\_\_);

}

else {

log(TAG "%s(): no connection found, report this error\n", \_\_func\_\_);

}

}

void CProtoSocket::handleRecv(SockPacket \*sockPkt)

{

SockDataHdr \*dataHdr = (SockDataHdr \*)sockPkt->data;

struct sockaddr\_in \*srcAddr = (struct sockaddr\_in \*)&dataHdr->srcAddr;

struct sockaddr\_in \*dstAddr = (struct sockaddr\_in \*)&dataHdr->dstAddr;

string key = CTCP::instance()->keyOf(srcAddr->sin\_addr, srcAddr->sin\_port, dstAddr->sin\_addr, dstAddr->sin\_port);

// find connection first

log(TAG "%s(): %s\n", \_\_func\_\_, key.c\_str());

ConnPMap::iterator it = \_connPPool.find(key);

if (it == \_connPPool.end()) {

log(TAG "%s(): no connection found, report this error\n", \_\_func\_\_);

return ;

}

SockDataHdr sdh = \*dataHdr;

sdh.srcAddr = dataHdr->dstAddr;

sdh.dstAddr = dataHdr->srcAddr;

sdh.flag = 0;

InetConnSock \*ics = it->second;

if (ics->recvQueue.empty()) {

sdh.len = -1;

}

else {

char \*pData = \_pBlock->buf1;

pData += sizeof(SockDataHdr);

// todo: copy recvQueue data to buffer

int goalLen = dataHdr->len;

int copiedLen = 0;

PacketQueue & recvQueue = ics->recvQueue;

while (copiedLen < goalLen) {

std::shared\_ptr<packet\_t> &ppkt = recvQueue.front();

int dataLen = ppkt->len;

if (dataLen > goalLen - copiedLen) {

dataLen = goalLen - copiedLen;

}

memcpy(pData, ppkt->data, dataLen);

pData += dataLen;

copiedLen += dataLen;

ppkt->pull(dataLen);

if (ppkt->empty()) {

recvQueue.pop\_front();

if (recvQueue.empty())

break;

}

}

sdh.len = copiedLen;

}

memcpy(\_pBlock->buf1, &sdh, sizeof(SockDataHdr));

afterHandle(ics->ics\_pid, SIGUSR2, \_\_func\_\_);

// todo: notify TCP we received data

}

void CProtoSocket::handleClose(SockPacket \*sockPkt)

{

Sock \*sock = (Sock \*)sockPkt->data;

log(TAG "%s() : close socket %d, port %d\n", \_\_func\_\_, sock->sockfd, ntohs(sock->port));

if (sock->type == SOCK\_STREAM) {

// need do 4wwh

string name = CTCP::keyOf(sock->addr, sock->port, sock->peerAddr, sock->peerPort);

ConnPMap::iterator it = \_connPPool.find(name);

if (it != \_connPPool.end()) {

if (it->second->\_inetSock.\_sock.state == SS\_DISCONNECTING) {

log(TAG "%s(): connection is closing...\n", \_\_func\_\_);

}

else {

it->second->\_inetSock.\_sock.state = SS\_DISCONNECTING;

CTCP::instance()->close(name);

}

}

else {

log(TAG "%s(): no connection find '%s'\n", \_\_func\_\_, name.c\_str());

afterHandle(0, sock->pid, SIGUSR1, \_\_func\_\_);

}

}

else if (sock->type == SOCK\_DGRAM) {

\_sockPool.erase(sock->sockfd);

afterHandle(1, sock->pid, SIGUSR1, \_\_func\_\_);

}

else {

log (TAG "%s(): unsupport sock type: %d\n", \_\_func\_\_, sock->type);

afterHandle(0, sock->pid, SIGUSR1, \_\_func\_\_);

}

}

void CProtoSocket::handleListen(SockPacket \*sockPkt)

{

log(TAG "%s().\n", \_\_func\_\_);

Sock \*sock = (Sock \*)sockPkt->data;

InetSock &cached = \_sockPool.at(sock->sockfd);

char \*pData = sockPkt->data;

pData += sizeof(Sock);

int backlog = \*(int \*)pData;

cached.backlog = backlog;

cached.sk\_state = LISTEN;

cached.\_sock.state = SS\_CONNECTING; // optional

CTCP::instance()->listen(&cached);

afterHandle(1, cached.sk\_pid, SIGUSR1, \_\_func\_\_);

}

void CProtoSocket::handleConnect(SockPacket \*sockPkt)

{

Sock \*sock = (Sock \*)sockPkt->data;

InetSock &cached = \_sockPool.at(sock->sockfd);

cached.sk\_peerAddr = sock->peerAddr;

cached.sk\_peerPort = sock->peerPort;

// set local address as needed

setLocalAddr(&cached);

int result = cached.\_sock.state;

if (result == SS\_UNCONNECTED) {

cached.\_sock.state = SS\_CONNECTING;

CTCP::instance()->connect(&cached);

// if connect successfully, connectFinished() is called

}

else {

log(TAG "Not unconnected socket: %d.\n", result);

memcpy(\_pBlock->buf1, &result, sizeof(result));

afterHandle(cached.sk\_pid, SIGUSR1, \_\_func\_\_);

}

}

void CProtoSocket::handleAccept(SockPacket \*sockPkt)

{

log(TAG "%s().\n", \_\_func\_\_);

Sock \*sock = (Sock \*)sockPkt->data;

// when there is a connected connection, return it, otherwise, record an accept request

//

// find a connection without sockfd assigned

//

ConnPMap::iterator it = std::find\_if(\_connPPool.begin(), \_connPPool.end(), [=](const ConnPMap::value\_type &pair){

InetConnSock \*conn = pair.second;

return conn->ics\_sockfd == 0 && conn->ics\_port == sock->port;

});

if (it != \_connPPool.end()) {

// find a connection, return it

InetConnSock \*ics = it->second;

ics->ics\_sockfd = selectFD();

Sock \*newSock = (Sock \*)ics;

memcpy(\_pBlock->buf1, newSock, sizeof(Sock));

afterHandle(newSock->pid, SIGUSR1, \_\_func\_\_);

}

else {

// no available connection yet

\_pendingAccept.insert(sock->port);

}

}

void CProtoSocket::connectFinished(string name, InetConnSock \*ics)

{

log(TAG "%s(): %s.\n", \_\_func\_\_, name.c\_str());

ics->\_inetSock.\_sock.state = SS\_CONNECTED;

\_connPPool.emplace(name, ics);

// here, no notify the connected address, a data structure must be returned instead of

// a single flag show failed or success

memcpy(\_pBlock->buf1, ics, sizeof(Sock));

afterHandle(ics->ics\_pid, SIGUSR1, \_\_func\_\_);

}

void CProtoSocket::accepted(std::string name, InetConnSock \*ics)

{

log(TAG "%s(): %s.\n", \_\_func\_\_, name.c\_str());

ics->\_inetSock.\_sock.state = SS\_CONNECTED;

auto pair = \_connPPool.emplace(name, ics);

std::set<uint16\_t>::iterator it = \_pendingAccept.find(ics->ics\_port);

if (it != \_pendingAccept.end()) {

ics->ics\_sockfd = selectFD();

memcpy(\_pBlock->buf1, ics, sizeof(Sock));

\_pendingAccept.erase(ics->ics\_port);

afterHandle(ics->ics\_pid, SIGUSR1, \_\_func\_\_);

}

else {

log (TAG "%s(): no accept request at port %d\n", \_\_func\_\_, ics->ics\_port);

}

}

void CProtoSocket::closed(std::string name)

{

log(TAG "%s(): %s\n", \_\_func\_\_, name.c\_str());

// remove connection

ConnPMap::iterator it = \_connPPool.find(name);

int result;

uint32\_t pid = it->second->ics\_pid;

if (it != \_connPPool.end()) {

\_connPPool.erase(it);

log(TAG "%s(): now there is %d connection\n", \_\_func\_\_, \_connPPool.size());

result = 1;

}

else {

result = 0;

log(TAG "%s(): connection not found\n", \_\_func\_\_);

}

log(TAG "%s(): now there is %d socket\n", \_\_func\_\_, \_sockPool.size());

afterHandle(result, pid, SIGUSR1, \_\_func\_\_);

}

uint16\_t CProtoSocket::selectPort()

{

return 1314;

}

uint32\_t CProtoSocket::selectFD()

{

return 15110;

}

void CProtoSocket::setLocalAddr(InetSock \* sock)

{

// get source ip address if not bind yet,

// if has bound, port will not be 0

if (sock->sk\_port == 0) { // not bind yet

const Device \*dev = CHardware::instance()->getDefaultDevice();

sock->sk\_addr = dev->ipAddr;

sock->sk\_port = htons(selectPort());

} else {}

}

void CProtoSocket::bytesAvailable(InetConnSock \*ics)

{

log (TAG "%s()\n", \_\_func\_\_);

}

void CProtoSocket::received(const packet\_t \*pkt)

{

log (TAG "Received %d bytes data.\n", pkt->len);

log (TAG "\_pendSocks: \n");

for\_each (\_pendingSocks.cbegin(), \_pendingSocks.cend(), [=](const InetSock \*sock){

log("pid: %d, sockfd: %d, port: %d\n", sock->sk\_pid, sock->sk\_sockfd, ntohs(sock->sk\_port));

});

// find pending socket

auto p = std::find\_if(\_pendingSocks.cbegin(), \_pendingSocks.cend(),

[=](const InetSock\* sock){

return (sock->sk\_port == pkt->dport);

});

if (p != \_pendingSocks.cend()) {

const InetSock\* sock = \*p;

// todo: copy pkt data to shared memory

SockDataHdr dataHdr;

dataHdr.sockfd = sock->sk\_sockfd;

dataHdr.len = pkt->len;

struct sockaddr\_in srcAddr;

srcAddr.sin\_addr = pkt->saddr;

srcAddr.sin\_port = pkt->sport;

srcAddr.sin\_family = AF\_INET;

struct sockaddr\_in dstAddr;

dstAddr.sin\_addr = pkt->daddr;

dstAddr.sin\_port = pkt->dport;

dstAddr.sin\_family = AF\_INET;

dataHdr.srcAddr = srcAddr;

dataHdr.dstAddr = dstAddr;

dataHdr.flag = 0;

char \*pData = \_pBlock->buf1;

memcpy(pData, &dataHdr, sizeof(dataHdr));

pData += sizeof(dataHdr);

memcpy(pData, pkt->data, pkt->len);

pData += pkt->len;

\_pendingSocks.erase(p);

afterHandle(sock->sk\_pid, SIGUSR2, \_\_func\_\_);

}

else {

log(TAG "No pending socket port %d find.\n", ntohs(pkt->dport));

}

}

void CProtoSocket::afterHandle(int success, int pid, int signo, const char \* const funcName)

{

memcpy(\_pBlock->buf1, &success, sizeof(success));

afterHandle(pid, signo, funcName);

}

void CProtoSocket::afterHandle(int pid, int signo, const char \*funcName)

{

usleep(100); // VIP: wait CSocket enter pause() statement

kill(pid, signo);

//log(TAG "%s() : kill signal %d to process %d.\n", funcName, signo, pid);

}

#include "CheckSum.h"

uint16\_t cksum(const u\_char \*const buf, size\_t size)

{

uint32\_t sum;

uint16\_t \*p = (uint16\_t \*)buf;

sum = 0;

while(size > 1) {

sum += \*p++;

size -= 2;

}

// padding as needed

if(size == 1) {

sum += \*((u\_char \*)p);

}

while(sum >> 16)

sum = (sum & 0xFFFF) + (sum >> 16);

return (uint16\_t)((~sum) & 0xFFFF);

}