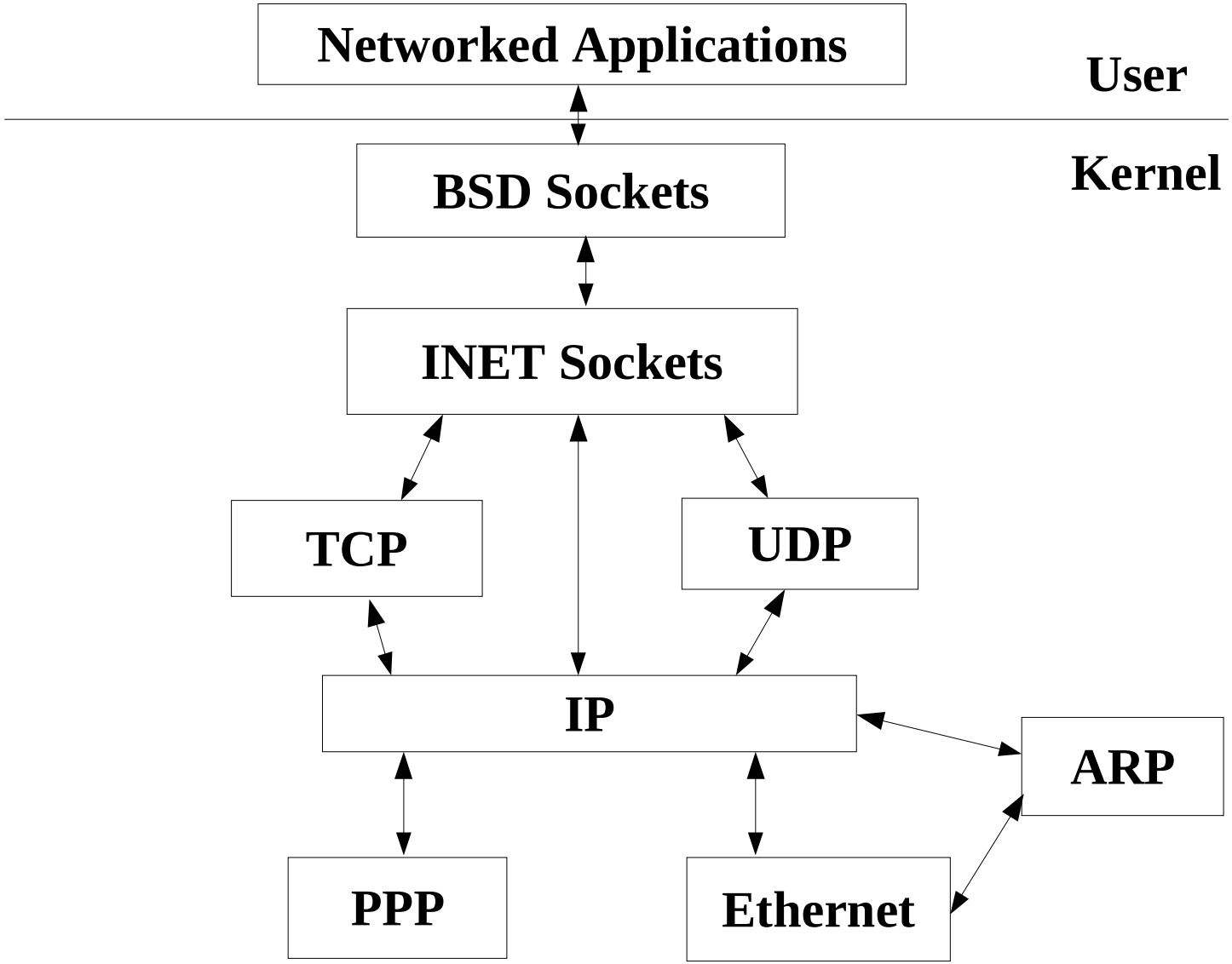
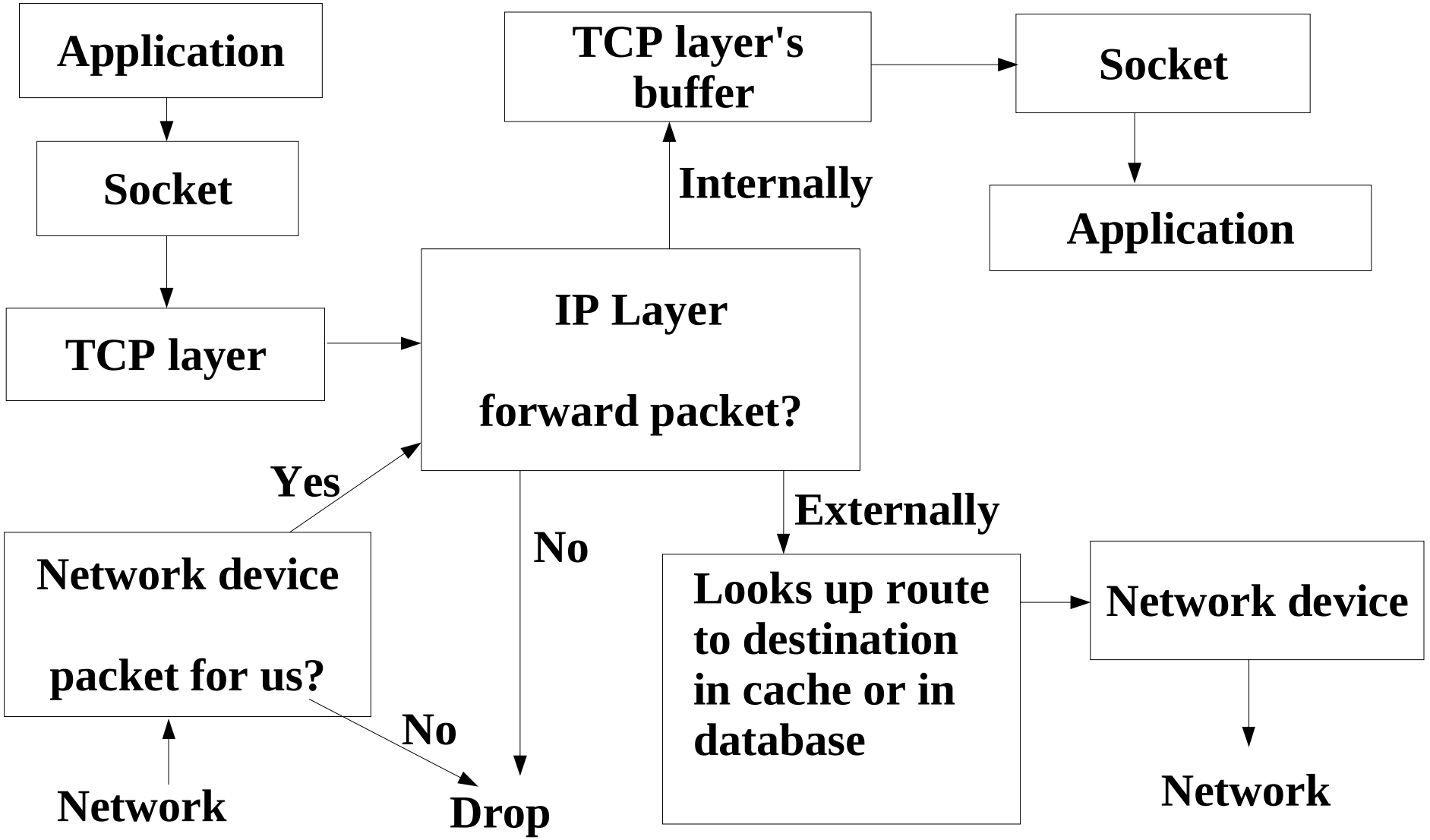
TCP/IP Stack in Linux

- **BSD Sockets**
- **INET Sockets**
- **TCP or UDP - transport layer**
- **IP - network layer**
- **Network interfaces representing network devices (Ethernet, PPP, etc) - link layer**
- **Network device drivers**

Networking Layers



Route of message in the stack



Architecture and Coding

- **modular, layered and object-oriented**
- **uses lot of function pointers to achieve that**
- **usage of one communication protocol and network device by socket layers is enabled assigning correct function pointers to items of various struct during system or connection initialization**
- **code isn't very well documented and therefore not very understandable clever desing makes it hard to follow call-paths**
- **easy to see that some parts of code were inserted of modified long time after the original implementation**
  - **e.g. making two functions from one, where the second has the same name with '2' at the end, etc**

struct sk_buff

- see include/linux/skbuff.h
- stores individual packets (message + headers)
- common for all protocols
- maintains strict layering of protocols without copying payload and headers all the time
- contains pointer to attached memory for data and lot of entries for simple access to headers
- protocol layers gradually add headers to both ends of packet content
- lot of library routines is provided for work with socket buffers

struct sk_buff (2)

- each packet corresponds to one sk_buff
- sk_buff is allocated big enough to store message data and all headers
- data are copied right after initialization
- protocol headers are inserted before or after current payload as the message goes through all layers (TCP, IP, Ethernet)
- complete packet can't be bigger than originally allocated sk_buff
- headers are added to sk_buff->data
- pointer to the header is stored in sk_buff too

struct sk_buff (3)

- only content of sk_buff->data is sent
- packet payload is copied only twice
  - once between user and kernel space
  - once between kernel space and output medium
- copy between user and kernel is done in TCP/UDP layer

BSD Sockets

- **struct socket - basis for implementation of BSD sockets, initialized in socket() call, list of important entries follows**
- **short type - values SOCK_DGRAM or SOCK_STREAM**
- **struct proto_ops *ops - TCP/UDP operations for the socket (bind, connect, accept, etc)**
- **struct inode *inode - associated file inode**
- **struct sock *sk - associated INET socket**
- **struct file *file - contains pointer to functions for write() and read()**

INET Sockets

- **struct sock - for network-oriented parts of sockets, important entries follows**
- **struct sock *next,*prev - socks are in linked list**
- **struct socket *socket - parent BSD socket**
- **__u32 daddr - destination IP address**
- **__u16 dport - destination IP port**
- **__u32 rcv_saddr - source IP address**
- **unsigned short num - local port**
- **struct proto *prot - TCP/UDP specific operations**

INET Sockets (2)

- **struct dst_entry *dst_cache - pointer to the route cache entry used by the socket**
- **struct sk_buff_head *receive_queue - head of the receive queue**
- **struct sk_buff_head *write_queue - head of the send queue**
- **see include/net/sock.h**
- **BSD socket has AF_INET, SOCK_STREAM and TCP's proto_ops**
- **INET socket has receive queue, write queue and lots of other TCP data**

IP Layer

- looks up route to the destination in routing cache
- uses Forwarding Information Base (FIB) if a search in the routing cache wasn't successful
  - entry found in FIB is stored in routing cache first and then used by IP layer
- only packets for local host are handed to TCP/UDP and higher layers
- routing cache is hash table containing up to 256 most recent routing entries
  - faster and smaller than FIB
  - filled with data from FIB
  - see /proc/net/rt_cache for its content

IP Layer (2)

- FIB is primary routing reference
  - keeps track of all known routes and destinations
  - filled at networking init with 'route' program
- entries are stored in up to 32 zones
  - one zone for each bit in IP addres
  - seach zone contains entries that can be identified with certain number of bits
  - search starts at more specific zones; goes through the table until it finds a match (default route)
  - see /proc/net/route for its content
- entries in FIB are permanent (deleted on reboot)

IP Layer (3)

- **neighbour table contains address information for computers physically connected to the host**
- **for each neighbour stores device and protocol usable for connection to that computer**
- **updated with Address Resolution Protocol (ARP)**
- **entries gets deleted if not used for some time**
- **see references and kernel sources for details on routing tables and routines**

IP Layer (4)

- **IP forwarding is simply combination of receiving and sending without involvement of TCP/UDP and higher layers**
- **forwarding is performed in ip_forward(), which is called in ip_rcv_finish()**
  - **see net/ipv4/ip_forward.c**
- **packet is received by IP and checked for errors**
  - **route can be modified if necessary**
  - **packet can be fragmented if it is too big**
  - **TTL is decremented; packet is dropped if it's zero**

Network Subsystem Initialization

- **inet_init() initializes network subsystem**
- **walks through array of protocols and calls init routine for each one**
- **routines for socket creation for all protocols register itself in BSD Sockets**
  - **they give name of protocol and pointer to function (INET - inet_create, UNIX - unix_create)**
- **see net/ipv4/af_inet.c**

Protocols

- **struct net_proto contains name of protocol (INET, IPX, UNIX) and init functions (e.g. inet_proto_init)**
  - **see net/protocols.c**
- **struct net_proto_family contains two items**
  - **int family**
  - **int (*create)(struct socket *sock, int protocol)**
- **struct proto_ops contains pointers to various protocol-dependent routines (e.g. bind, listen, accept, ioctl, etc)**

Packet types

- ptype_base is head of the list of all supported packet types, which the link layer can accept and system is able to process
- initialization routine for each protocol (e.g. inet_proto_init) calls routines like ip_init(), tcp_init(), arp_init(), etc
- subprotocols which want to register packet type (IP, ARP), call dev_add_pack() in their init function and specify handlerfor received packets (e.g. ip_rcv())
  - this add packet types to ptype_base
- net_rx_action is therefore able to give packet to correct protocol

Connection initialization

- syscall socket() is implemented by sys_socket()
- sys_socket() calls sock_create(), which creates BSD socket with inode and calls create routine for given address family - inet_create() for INET sockets
  - see net/socket.c
- inet_create() assigns operations for correct protocol to socket and creates all queues
  - see net/ipv4/af_inet.c
- syscall connect() goes to protocol dependent routine (tcp_v4_connect or udp_connect)
  - both of them establish route to destination
  - tcp_v4_connect() creates virtual connection too
  - see net/ipv4/tcp_output.c and net/ipv4/udp.c

Sockets and Routing

- route for a socket is looked up only once - during connection init
  - tcp_v4_connect() calls ip_route_connect() and stores the result in sock->dst_cache
  - that entry is changed only if the route doesn't work any more (neighbour computer is down)
  - see net/ipv4/route.c

Connection closing

- **syscall close() is implemented by sock_close()**
- **sock_close() calls sock_release(), which removes socket from the inode list and calls inet_release()**
- **inet_release() is INET socket release routine**
  - **cleans all queues and calls protocol's close routine - tcp_v4_close() or udp_close()**
- **no change in routing cache is performed; entry is removed from the cache if it isn't used for some time or there is no free space left**

Transmission of data to network

- **syscall write() is implemented with sys_write()**
- **sys_write() does something like files_struct[fd]->file->f_op->write, which usually means call to sock_write**
  - **see fs/read_write.c**
- **sock_write() calls socki_lookup(), which returns struct socket; then creates and fills message header (msghdr); calls sock_sendmsg() at the end**
  - **see net/socket.c**

Transmission of data to network (2)

- **sock_sendmsg() calls inet_sendmsg() through a function pointer sock->ops->sendmsg**
- **inet_sendmsg() is used for INET sockets (address family AF_INET)**
  - **it performs some socket checking and calls sk->prot[tcp/udp]->sendmsg()**
  - **sk is of type struct sock**
  - **see net/ipv4/af_inet.c**

Transmission of data to network (3)

- tcp_sendmsg() allocates memory for sk_buff, initializes it and fills it with message data; call path goes through various tcp routines and then steps into tcp_transmit_skb()
  - see net/ipv4/tcp.c
- tcp_transmit_skb() creates TCP header, adds it to sk_buff and counts checksum; calls ip_queue_xmit()
  - see net/ipv4/tcp_output.c
- udp_sendmsg() calls ip_build_xmit() instead

Transmission of data to network (4)

- ip_queue_xmit() creates IP headers, looks up route, add IP checksum, fragments packet and calls ip_queue_xmit2() through some macros
- ip_queue_xmit2() calls ip_output() through skb->dst->output; then follows some minor IP routines - last of them calls ip_finish_output2()
- ip_finish_output2() calls directly dev_queue_xmit() in case of existing hardware header cache (hh_cache); in other case calls neigh_{connected | compat | resolv}_output(), which then call dev_queue_xmit()
  - see net/core/neighbour.c and net/ipv4/arp.c

Transmission of data to network (5)

- **dev_queue_xmit() is routine, which sends packets to network device through the function pointer net_device->hard_start_xmit()**
  - **that pointer is filled with device dependent transmission routine during init of a network device in some "probe"”function**
  - **it is used as a common mediator between IP layer and network device dependent routines**

Transmission of data from network

- packet is received and interrupt is raised
- interrupt handler in driver copies data from device into new sk_buff, then calls netif_rx()
- routine netif_rx() puts packet into queue and issues softirq for later execution
  - see net/dev/core.c
- net_rx_action() is softirq handler, which is invoked by scheduler; call graph then walks through various routines into netif_receive_skb(), which call pt_prev->func() [= ip_rcv()] for handing of a packet to all protocols associated with the device which received the packet

Transmission of data from network (2)

- ip_rcv() is main IP receive routine
  - checks packet for errors (invalid length, invalid checksum, etc)
  - defragments packet if necessary
  - calls ip_rcv_finish() at the end
  - see net/ipv4/ip_input.c
- ip_rcv_finish() calls skb->dst->input(skb) which is set to ip_local_deliver() or ip_forward()
- ip_local_deliver() calls tcp_v4_rcv() or udp_rcv() through list inet_protocol

Transmission of data from network (3)

- tcp_v4_rcv() calls tcp_v4_do_rcv()
- tcp_v4_do_rcv() checks all flags and header, sends ACK and calls tcp_rcv_established() in case of established connection or tcp_rcv_state_process() otherwise
  - both these routines are quite complex
- tcp_rcv_established() adds packet to socket receive queue for correct socket
- if the process owning the socket sleeps in read() then it is woken up

Transmission of data from network (4)

- **syscall read() is implemented with sys_read()**
- **sys_read() calls sock_read() through function pointer file->f_op->read**
  - **see fs/read_write.c**
- **sock_read() calls socki_lookup() and sock_recvmsg()**
  - **see net/socket.c**
- **sock_recvmsg() calls inet_recvmsg() through function pointer sock->ops->recvmsg**
  - **see net/socket.c**

Transmission of data from network (5)

- **inet_recvmsg() is used for INET sockets; it calls receive routine for correct protocol through pointer sk->prot->recvmsg - it is set to tcp_recvmsg() in case of TCP**
  - **see net/ipv4/af_inet.c**
- **tcp_recvmsg() reads message from socket receive queue if there is some; gets blocked otherwise**
  - **see net/ipv4/tcp.c**

Network devices (interfaces)

- network devices aren't represented by file in /dev directory because they "push" data
- each network device is represented by struct net_device
- all these structures are in linked list with dev_base as a head
  - it is static list, limits number of devices of each type (eth0 - eth7, but no more)
- example: dev_base -> lo0 -> eth0 (3Com card) -> eth1 (HP card)
- see include/linux/netdevice.h

Network device init

- network device hasn't /dev file so it must find out its name
- net_dev_init() is called during device init in register_netdevice() - it assigns handler net_rx_action to softirq NET_RX_SOFTIRQ
  - see net/core/dev.c
- list of devices is created during init of network subsystems with help of "probe" routines
- in case of modular driver is driver's probe executed at module load time - appends net_device at the end of list in case of success

struct net_device

- represents one network device, list of important entries follows
- char *name - for example "eth0"
- unsigned long base_addr - I/O address
- unsigned int irq - IRQ number
- unsigned mtu - largest packet size
- struct net_device *next
- int (*init)(struct net_device *dev)
- int (*hard_start_xmit)(struct sk_buff *skb, struct net_device *dev) - transmission routine
- see include/linux/netdevice.h

Ethernet interfaces

- ethif_probe() is called at init for each ethernet device (eth0..eth7) in the list of interfaces
- ethif_probe() calls probe_list() for each existing interface
- probe_list() tries to call "probe" routines specific for various device
  - ssuccess: assigns functions pointers from device driver to given struct net_device
  - failure: no more eth devices, remove structure from the list and return
- see drivers/net/Space.c

References

- **Linux IP Networking**
  - **http://kernelnewbies.org/documents/ipnetworking/linuxipnetworking.html**
- **Kernel sources (version 2.4.20)**
- **Linux Cross-Reference**
  - **http://lxr.linux.no**