Build Your Own Networking Stack

Pengcheng Xu
School of EECS, Peking University
December 7, 2019

“Computer Networking”

What comes to your mind FIRST?

“Computer Networking”

What do you NEED to build that?

Computer Networking

- What is **Internet**?
  - The Internet (portmanteau of interconnected network) is the global system of interconnected computer networks that use the Internet protocol suite (TCP/IP) to link devices worldwide.
- Interconnected
- Internet protocol suite
  - TCP
  - IP

Computer Networking, cont’d

- What is **Web**?
  - The World Wide Web (WWW), commonly known as the Web, is an information system where documents and other web resources are identified by Uniform Resource Locators, which may be interlinked by hypertext, and are accessible over the Internet.
- URLs
  - Host: DNS, IP
    - UDP
  - Port: TCP / QUIC
  - Application protocol (a.k.a. scheme): HTTP / HTTPS / WebSocket
  - resource path, parameters, …

Computer Networking, cont’d

- What we have now?
  - HTTP/HTTPS/WebSocket: application layer
    - Nginx, Apache, Curl, …
  - TCP/UDP/*QUIC*: transport layer
    - OS kernel (, DPDK, …)
  - IP: networking layer
    - OS kernel (, NIC, HCA, …)
  - Ethernet: link layer
    - OS kernel or NIC
  - PHY (+MAC, RRC, …): physical layer
    - NIC (, SDR, …)

What Fun Can We Have?

- Design a protocol?
  - Lacks interoperability with others
    - It's exciting to see things work, or "working with others"
  - Also, hard to get correct
    - RFC791 / RFC793 are published in '90s and matured for decades
- *Play with* a protocol?
  - Or, protocols?

Hack it!

- Application protocol?
  - Easy to hack
    - As it’s easy to write one your self (HTTP/1.1 via telnet)
  - Maybe not that interesting
- Transport layer / Networking Layer / Link Layer?
  - OS kernels are <u>hard</u> to hack
    - Tunable knobs are limited
    - Small code changes may break it all
- Physical layer?
  - Basically out of reach (signal processing, VLSI design, …)

Bypass it!

- Goal: to eliminate OS's interference
    - Turn things off
        - While maintaining ability to do things
- What do we have?
    - IPtables & iproute
        - Filter out all L3 packets (that came from kernel)
        - Turn off ARP
    - Linux raw sockets / TAP
        - May be affected by iptables
    - Pcap
        - Can inject packets too!

PKU CompNet’19 Lab 2

- Goal:
  - “In this lab, you will implement a userspace C/C++ program based on libpcap to replace layer 2/3/4 of the kernel protocol stack from scratch.”
- 3 parts in total:
  - Link-layer: Packet I/O On Ethernet
  - Network-layer: IP Protocol
    - Exchange of routing information between *your instances* REQUIRED
  - Transport-layer: TCP Protocol
    - LD_PRELOAD’able

Evaluation

- Application works
  - *Can select which application to test on*
- Other hosts not confused
  - E.g. routes correctly, talks correctly, …
  - Functionality test in netns, real world tests
- Works robustly
  - Netem
  - Unexpected termination?

Complete the Stack

- Ethernet – send / recv
  - Ethernet-II frame (checksums)
  - Multicasting & Broadcasting
- IP – send / recv
  - ARP
  - Routing table design (Longest Prefix Matching, Aging, Metrics)
  - Multicasting
  - Routing Information Protocol (RIP)
- UDP (for RIP & socket API testing)
  - Checksums
- TCP
  - Connection establishment / teardown
  - Reliable delivery
  - Flow control
- Socket API
  - FD partitioning (avoiding clash)

Pitfalls

- Kernel may discard packets that fail sanity checks
  - E.g. wrong Ethernet source address
- Kernel (when testing) may send non-conformant packets to you
  - Checksum offloading!
- User applications may be a jerk
  - Checking fd number *values*
  - Fork
  - Non-conformant use of sockets

My Design

- Client-Server model
  - Can support multiple applications
  - Linux abstract UNIX domain sockets
- Event-driven model
- Boost::Asio used
  - The major cause of troubles
- But still forged something like an event queue
  - Most significant design failure

Endianness!!!

Before starting out for details

Details – Client-server Model

- Client and server keep a .h in sync
  - Typed union in struct, with common fields
  - Common field for requests / responses:
    - Type
    - Client-id
    - Body length
  - Req/resp for read/write of all protocols plus socket API
  - If request/response has body, data directly follows
- Server places requests in queue, tagged with client-id
  - So we can cancel then when a client disconnects
- The domain socket works synchronously
  (with socket::read/write)

Details – Ethernet

- Libpcap provides a selectable fd:
  - pcap_get_selectable_fd
  - Create unix stream from fd in Boost::Asio, we have async_read/async_write now
- Packet headers hand-written
  - Linux has headers (for raw sockets), but naming is weird
- Async read frame / write frame
  - Callback via std::function (from lambdas)
    - Don’t capture via reference for callbacks!
  - Upper-layer protocols wrap handlers
- Asio async loop constantly captures frames
- Multicast / broadcast packets: magic addresses & default parameters

Details – Ethernet / IP

- Ingress frames are dispatched according to Ethertype
  - But, in a special way
- Core list of handlers (std::functions) for each read request
  - Handlers are called in turn
  - If handler consumes frame: return true
    - Handler gets removed from list
    - Frame consumed, finish
  - If handler does not consume frame: return false
    - Calls next handler in list
- Protocols register default handlers
- Read operation register disposable handlers

Details - IP

- Default ingress handler: ICMP echo reply
- Egress routine:
  - Resolve nexthop via routing table
  - Resolve nexthop MAC via ARP table
    - Possibly sends ARP requests / waits for responses
    - Possibly multicast, magic ethernet address
  - Write Ethernet frame
- Ingress routine:
  - User provide handler
  - Async_read function wrap handler to check Ethertype, checksum, …

Details – IP Routing

- Routing Information Protocol
  - Need UDP transport (an easy one)
  - Broadcast on enter, multicast on update
    - Broadcast/multicast for IP, Ethernet
  - Distributed Bellman-Ford
  - Routes age and get deleted in a 2-tier way
- Interoperable with real RIP agents (BIRD)!

Details – UDP / IP

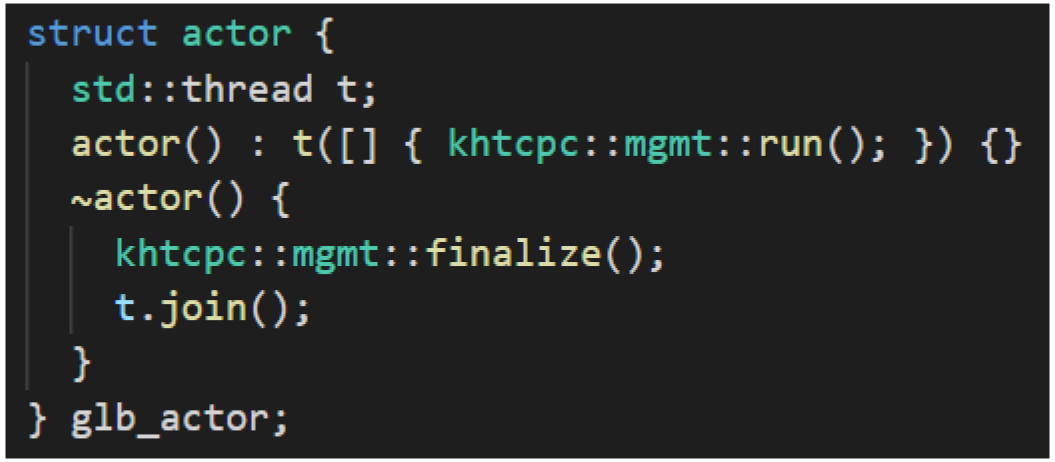
- No complications: unreliable datagram delivery
- Checksums:
  - Pseudo headers
  - Zero handling
  - Per standard, can be disabled (all-zero)

Details – Socket API

- Socket can be implemented once we have UDP
  - SOCK_DGRAM over INET
- We don’t want to mess with system fds
  - Provide fake fds
  - Split the fd number assignment
    - Some applications don’t like this
- Server only implements async sockets
  - Synchronous client waits on condition variable for async complete
- Getaddrinfo only does IP address resolving
  - DNS is complex

Details – Socket API, cont'd

- Extern "C" for LD_PRELOAD
  - Use the global struct ctor/dtor trick for global initialization / cleanup



```
struct actor {
  std::thread t;
  actor() : t([] { khtcpc::mgmt::run(); }) {}
  ~actor() {
    khtcpc::mgmt::finalize();
    t.join();
  }
} glb_actor;
```

Details - TCP

- Disclaimer: this section is not finished
  - Design & implementation failures
  - Only handshake implemented
- Segment sending / receiving
- Connection state machines
  - Via callback injections
    - Callback injects (calls) other callback according to state
- Connection control blocks
  - Stored in map with 4-tuple+client_id as key
    - Tagged with client id for teardown on client disconnect
  - Sequence / Acknowledgement numbers

Details – TCP, cont’d

- Flow control (should be done this way)
    - Maintain 5 numbers:
        - Send-Unack
        - Send-Next
        - Send-Until
        - Recv-Next
        - Recv-Window
- For simplicity:
    - Back-off-1: send segment, expect ack
    - Low performance, but it works

Why I Failed

- My forged “event loop” (the handler list) does not support timeouts
    - Can only cancel by Boost::Asio timers and removing handler from queue
    - Slow, unreliable, complicated
    - Should have used a good event loop (e.g. libuv)
- C++ lambda constructs are crappy and difficult to use
    - Memory problems from reference
    - Performance & coherency problems from copying
    - Lambda functions render call stacks unreadable

Live Demonstration

About the Course

- 04832250: Computer Networks (Honor Track)
  - By Prof. Chenren Xu, PKU
  - Traditional topics + Wireless + paper reading + research project
- 04830241: Computer Networks Practicum
  - 3 Labs:
    1. Cross layer analysis over LTE and WLAN networks
    2. Implement your own TCP/IP stack
    3. Multipath HTTP/2

Thank You